DAVID HAREL, DEXTER KOZEN, AND JERZY TIURYN

DYNAMIC LOGIC

PREFACE

Dynamic Logic (DL) is a formal system for reasoning about programs. Traditionally, this has meant formalizing correctness specifications and proving rigorously that those specifications are met by a particular program. Other activities fall into this category as well: determining the equivalence of programs, comparing the expressive power of various programming constructs, synthesizing programs from specifications, *etc*. Formal systems too numerous to mention have been proposed for these purposes, each with its own peculiarities.

DL can be described as a blend of three complementary classical ingredients: first-order predicate logic, modal logic, and the algebra of regular events. These components merge to form a system of remarkable unity that is theoretically rich as well as practical.

The name *Dynamic Logic* emphasizes the principal feature distinguishing it from classical predicate logic. In the latter, truth is *static*: the truth value of a formula φ is determined by a valuation of its free variables over some structure. The valuation and the truth value of φ it induces are regarded as immutable; there is no formalism relating them to any other valuations or truth values. In Dynamic Logic, there are explicit syntactic constructs called *programs* whose main role is to change the values of variables, thereby changing the truth values of formulas. For example, the program $x := x + 1$ over the natural numbers changes the truth value of the formula "x is even".

Such changes occur on a metalogical level in classical predicate logic. For example, in Tarski's definition of truth of a formula, if $u : \{x, y, \dots\} \to \mathbb{N}$ is a valuation of variables over the natural numbers $\mathbb{N}$, then the formula $\exists x\ x^2 = y$ is defined to be true under the valuation u iff there exists an $a \in \mathbb{N}$ such that the formula $x^2 = y$ is true under the valuation $u[x/a]$, where $u[x/a]$ agrees with u everywhere except x, on which it takes the value a. This definition involves a metalogical operation that produces $u[x/a]$ from u for all possible values $a \in \mathbb{N}$. This operation becomes explicit in DL in the form of the program $x := ?$, called a *nondeterministic* or *wildcard assignment*. This is a rather unconventional program, since it is not effective; however, it is quite useful as a descriptive tool. A more conventional way to obtain a square root of y, if it exists, would be the program

$$x := 0\ ;\ \textbf{while}\ x^2 < y\ \textbf{do}\ x := x + 1. \tag{1}$$

In DL, such programs are first-class objects on a par with formulas, complete with a collection of operators for forming compound programs inductively from a basis of primitive programs. To discuss the effect of the execution of a program α on the truth of a formula φ, DL uses a modal construct $<\alpha>\varphi$, which intuitively states, "It is possible to execute α starting from the current state and halt in a state satisfying φ." There is also the dual construct $[\alpha]\,\varphi$, which intuitively states, "If α halts when started in the current state, then it does so in a state satisfying φ." For example, the first-order formula $\exists x\ x^2 = y$ is equivalent to the DL formula $<x := ?>\ x^2 = y$. In order to instantiate the quantifier effectively, we might replace the nondeterministic assignment inside the < > with the **while** program (1); over $\mathbb{N}$, the two formulas would be equivalent.

Apart from the obvious heavy reliance on classical logic, computability theory and programming, the subject has its roots in the work of [Thiele, 1966] and [Engeler, 1967] in the late 1960's, who were the first to advance the idea of formulating and investigating formal systems dealing with properties of programs in an abstract setting. Research in program verification flourished thereafter with the work of many researchers, notably [Floyd, 1967], [Hoare, 1969], [Manna, 1974], and [Salwicki, 1970]. The first precise development of a DL-like system was carried out by [Salwicki, 1970], following [Engeler, 1967]. This system was called Algorithmic Logic. A similar system, called Monadic Programming Logic, was developed by [Constable, 1977]. Dynamic Logic, which emphasizes the modal nature of the program/assertion interaction, was introduced by [Pratt, 1976].

Background material on mathematical logic, computability, formal languages and automata, and program verification can be found in [Shoenfield, 1967] (logic), [Rogers, 1967] (recursion theory), [Kozen, 1997a] (formal languages, automata, and computability), [Keisler, 1971] (infinitary logic), [Manna, 1974] (program verification), and [Harel, 1992; Lewis and Papadimitriou, 1981; Davis *et al.*, 1994] (computability and complexity). Much of this introductory material as it pertains to DL can be found in the authors' text [Harel *et al.*, 2000].

There are by now a number of books and survey papers treating logics of programs, program verification, and Dynamic Logic [Apt and Olderog, 1991; Backhouse, 1986; Harel, 1979; Harel, 1984; Parikh, 1981; Goldblatt, 1982; Goldblatt, 1987; Knijnenburg, 1988; Cousot, 1990; Emerson, 1990; Kozen and Tiuryn, 1990]. In particular, much of this chapter is an abbreviated summary of material from the authors' text [Harel *et al.*, 2000], to which we refer the reader for a more complete treatment. Full proofs of many of the theorems cited in this chapter can be found there, as well as extensive introductory material on logic and complexity along with numerous examples and exercises.

1 REASONING ABOUT PROGRAMS

1.1 Programs

For us, a *program* is a recipe written in a formal language for computing desired output data from given input data.

EXAMPLE 1. The following program implements the Euclidean algorithm for calculating the greatest common divisor (gcd) of two integers. It takes as input a pair of integers in variables x and y and outputs their gcd in variable x:

while $y \neq 0$ **do**
  begin
    $z := x \bmod y;$
    $x := y;$
    $y := z$
  end

The value of the expression $x \bmod y$ is the (nonnegative) remainder obtained when dividing x by y using ordinary integer division.

Programs normally use *variables* to hold input and output values and intermediate results. Each variable can assume values from a specific *domain of computation*, which is a structure consisting of a set of data values along with certain distinguished constants, basic operations, and tests that can be performed on those values, as in classical first-order logic. In the program above, the domain of x, y, and z might be the integers $\mathbb{Z}$ along with basic operations including integer division with remainder and tests including $\neq$. In contrast with the usual use of variables in mathematics, a variable in a program normally assumes different values during the course of the computation. The value of a variable x may change whenever an assignment $x := t$ is performed with x on the left-hand side.

In order to make these notions precise, we will have to specify the programming language and its semantics in a mathematically rigorous way. In this section we give a brief introduction to some of these languages and the role they play in program verification.

1.2 States and Executions

As mentioned above, a program can change the values of variables as it runs. However, if we could freeze time at some instant during the execution of the program, we could presumably read the values of the variables at that instant, and that would give us an instantaneous snapshot of all information that we would need to determine how the computation would proceed from that point. This leads to the concept of a *state*—intuitively, an instantaneous description of reality.

Formally, we will define a *state* to be a function that assigns a value to each program variable. The value for variable x must belong to the domain associated with x. In logic, such a function is called a *valuation*. At any given instant in time during its execution, the program is thought to be "in" some state, determined by the instantaneous values of all its variables. If an assignment statement is executed, say $x := 2$, then the state changes to a new state in which the new value of x is 2 and the values of all other variables are the same as they were before. We assume that this change takes place instantaneously; note that this is a mathematical abstraction, since in reality basic operations take some time to execute.

A typical state for the gcd program above is $(15, 27, 0, \dots)$, where (say) the first, second, and third components of the sequence denote the values assigned to x, y, and z respectively. The ellipsis "$\dots$" refers to the values of the other variables, which we do not care about, since they do not occur in the program.

A program can be viewed as a transformation on states. Given an initial (input) state, the program will go through a series of intermediate states, perhaps eventually halting in a final (output) state. A sequence of states that can occur from the execution of a program α starting from a particular input state is called a *trace*. As a typical example of a trace for the program above, consider the initial state $(15, 27, 0)$ (we suppress the ellipsis). The program goes through the following sequence of states:

$$(15, 27, 0),\ (15, 27, 15),\ (27, 27, 15),\ (27, 15, 15),\ (27, 15, 12),\ (15, 15, 12),$$
$$(15, 12, 12),\ (15, 12, 3),\ (12, 12, 3),\ (12, 3, 3),\ (12, 3, 0),\ (3, 3, 0),\ (3, 0, 0).$$

The value of x in the last (output) state is 3, the gcd of 15 and 27.

The binary relation consisting of the set of all pairs of the form (input state, output state) that can occur from the execution of a program α, or in other words, the set of all first and last states of traces of α, is called the *input/output relation* of α. For example, the pair $((15, 27, 0), (3, 0, 0))$ is a member of the input/output relation of the gcd program above, as is the pair $((-6, -4, 303), (2, 0, 0))$. The values of other variables besides x, y, and z are not changed by the program. These values are therefore the same in the output state as in the input state. In this example, we may think of the variables x and y as the *input variables*, x as the *output variable*, and z as a *work variable*, although formally there is no distinction between any of the variables, including the ones not occurring in the program.

1.3 Programming Constructs

In subsequent sections we will consider a number of programming constructs. In this section we introduce some of these constructs and define a few general classes of languages built on them.

In general, programs are built inductively from *atomic programs* and *tests* using various *program operators*.

While *Programs*

A popular choice of programming language in the literature on DL is the family of deterministic **while** programs. This language is a natural abstraction of familiar imperative programming languages such as Pascal or C. Different versions can be defined depending on the choice of tests allowed and whether or not nondeterminism is permitted.

The language of **while** programs is defined inductively. There are atomic programs and atomic tests, as well as program constructs for forming compound programs from simpler ones.

In the propositional version of Dynamic Logic (PDL), atomic programs are simply letters $a, b, \ldots$ from some alphabet. Thus PDL abstracts away from the nature of the domain of computation and studies the pure interaction between programs and propositions. For the first-order versions of DL, atomic programs are *simple assignments* $x := t$, where x is a variable and t is a term. In addition, a *nondeterministic* or *wildcard assignment* $x := ?$ or *nondeterministic choice* construct may be allowed.

Tests can be *atomic tests*, which for propositional versions are simply propositional letters p, and for first-order versions are atomic formulas $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and p is an n-ary relation symbol in the vocabulary of the domain of computation. In addition, we include the *constant tests* **1** and **0**. Boolean combinations of atomic tests are often allowed, although this adds no expressive power. These versions of DL are called *poor test*.

More complicated tests can also be included. These versions of DL are sometimes called *rich test*. In rich test versions, the families of programs and tests are defined by mutual induction.

Compound programs are formed from the atomic programs and tests by induction, using the *composition*, *conditional*, and *while* operators. Formally, if φ is a test and α and β are programs, then the following are programs:

- $\alpha\ ;\ \beta$
- **if** φ **then** α **else** β
- **while** φ **do** α.

We can also parenthesize with **begin** ... **end** where necessary. The gcd program of Example 1 above is an example of a **while** program.

The semantics of these constructs is defined to correspond to the ordinary operational semantics familiar from common programming languages.

Regular Programs

Regular programs are more general than **while** programs, but not by much. The advantage of regular programs is that they reduce the relatively more complicated

while program operators to much simpler constructs. The deductive system becomes comparatively simpler too. They also incorporate a simple form of nondeterminism.

For a given set of atomic programs and tests, the set of *regular programs* is defined as follows:

(i) any atomic program is a program

(ii) if φ is a test, then $\varphi?$ is a program

(iii) if α and β are programs, then $\alpha\ ;\ \beta$ is a program;

(iv) if α and β are programs, then $\alpha \cup \beta$ is a program;

(v) if α is a program, then α^* is a program.

These constructs have the following intuitive meaning:

(i) Atomic programs are basic and indivisible; they execute in a single step. They are called *atomic* because they cannot be decomposed further.

(ii) The program $\varphi?$ tests whether the property φ holds in the current state. If so, it continues without changing state. If not, it blocks without halting.

(iii) The operator ; is the *sequential composition* operator. The program $\alpha\ ;\ \beta$ means, "Do α, then do β."

(iv) The operator $\cup$ is the *nondeterministic choice* operator. The program $\alpha \cup \beta$ means, "Nondeterministically choose one of α or β and execute it."

(v) The operator * is the *iteration* operator. The program α means, "Execute α some nondeterministically chosen finite number of times."

Keep in mind that these descriptions are meant only as intuitive aids. A formal semantics will be given in Section 2.2, in which programs will be interpreted as binary input/output relations and the programming constructs above as operators on binary relations.

The operators $\cup$, ;, * may be familiar from automata and formal language theory (see [Kozen, 1997a]), where they are interpreted as operators on sets of strings over a finite alphabet. The language-theoretic and relation-theoretic semantics share much in common; in fact, they have the same equational theory, as shown in [Kozen, 1994a].

The operators of deterministic **while** programs can be defined in terms of the regular operators:

$$\mathbf{if}\ \varphi\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta \stackrel{\mathrm{def}}{=} \varphi?\ ;\ \alpha \cup \neg\varphi?\ ;\ \beta \tag{2}$$

$$\mathbf{while}\ \varphi\ \mathbf{do}\ \alpha \stackrel{\mathrm{def}}{=} (\varphi?\ ;\ \alpha)^*\ ;\ \neg\varphi? \tag{3}$$

The class of **while** programs is equivalent to the subclass of the regular programs in which the program operators $\cup$, ?, and * are constrained to appear only in these forms.

Recursion

Recursion can appear in programming languages in several forms. Two such manifestations are *recursive calls* and *stacks*. Under certain very general conditions, the two constructs can simulate each other. It can also be shown that recursive programs and **while** programs are equally expressive over the natural numbers, whereas over arbitrary domains, **while** programs are strictly weaker. **While** programs correspond to what is often called *tail recursion* or *iteration*.

R.E. Programs

A *finite computation sequence* of a program α, or *seq* for short, is a finite-length string of atomic programs and tests representing a possible sequence of atomic steps that can occur in a halting execution of α. Seqs are denoted $\sigma, \tau, \ldots$. The set of all seqs of a program α is denoted $CS(\alpha)$. We use the word "possible" loosely—$CS(\alpha)$ is determined by the syntax of α alone. Because of tests that evaluate to false, $CS(\alpha)$ may contain seqs that are never executed under any interpretation.

The set $CS(\alpha)$ is a subset of A^*, where A is the set of atomic programs and tests occurring in α. For **while** programs, regular programs, or recursive programs, we can define the set $CS(\alpha)$ formally by induction on syntax. For example, for regular programs,

$$\begin{aligned}
CS(a) &\stackrel{\text{def}}{=} \{a\}, \quad a \text{ an atomic program or test} \\
CS(\mathbf{skip}) &\stackrel{\text{def}}{=} \{\varepsilon\} \\
CS(\mathbf{fail}) &\stackrel{\text{def}}{=} \varnothing \\
CS(\alpha\,;\,\beta) &\stackrel{\text{def}}{=} \{\sigma\,;\,\tau \mid \sigma \in CS(\alpha),\ \tau \in CS(\beta)\} \\
CS(\alpha \cup \beta) &\stackrel{\text{def}}{=} CS(\alpha) \cup CS(\beta) \\
CS(\alpha^*) &\stackrel{\text{def}}{=} CS(\alpha)^* \\
&= \bigcup_{n \geq 0} CS(\alpha^n),
\end{aligned}$$

where

$$\begin{aligned}
\alpha^0 &\stackrel{\text{def}}{=} \mathbf{skip} \\
\alpha^{n+1} &\stackrel{\text{def}}{=} \alpha^n\,;\,\alpha.
\end{aligned}$$

For example, if a is an atomic program and p an atomic formula, then the program

$$\textbf{while}\ p\ \textbf{do}\ a \quad = \quad (p?\,;\,a)^*\,;\,\neg p?$$

has as seqs all strings of the form

$$(p?\,;\,a)^n\,;\,\neg p? \quad = \quad \underbrace{p?; a; p?; a; \cdots; p?; a}_{n}; \neg p?$$

for all $n \geq 0$. Note that each seq σ of a program α is itself a program, and

$$CS(\sigma) = \{\sigma\}.$$

While programs and regular programs give rise to regular sets of seqs, and recursive programs give rise to context-free sets of seqs. Taking this a step further, we can define an *r.e. program* to be simply a recursively enumerable set of seqs. This is the most general programming language we will consider in the context of DL; it subsumes all the others in expressive power.

Nondeterminism

We should say a few words about the concept of *nondeterminism* and its role in the study of logics and languages, since this concept often presents difficulty the first time it is encountered.

In some programming languages we will consider, the traces of a program need not be uniquely determined by their start states. When this is possible, we say that the program is *nondeterministic*. A nondeterministic program can have both divergent and convergent traces starting from the same input state, and for such programs it does not make sense to say that the program halts on a certain input state or that it loops on a certain input state; there may be different computations starting from the same input state that do each.

There are several concrete ways nondeterminism can enter into programs. One construct is the *nondeterministic* or *wildcard assignment* $x := ?$. Intuitively, this operation assigns an arbitrary element of the domain to the variable x, but it is not determined which one.[1] Another source of nondeterminism is the unconstrained use of the choice operator $\cup$ in regular programs. A third source is the iteration operator * in regular programs. A fourth source is r.e. programs, which are just r.e. sets of seqs; initially, the seq to execute is chosen nondeterministically. For example, over $\mathbb{N}$, the r.e. program

$$\{x := n \mid n \geq 0\}$$

is equivalent to the regular program

$$x := 0\,;\, (x := x + 1)^*.$$

Nondeterministic programs provide no explicit mechanism for resolving the nondeterminism. That is, there is no way to determine which of many possible next steps will be taken from a given state. This is hardly realistic. So why study nondeterminism at all if it does not correspond to anything operational? One good answer is that nondeterminism is a valuable tool that helps us understand the expressiveness of programming language constructs. It is useful in situations in

[1]This construct is often called *random assignment* in the literature. This terminology is misleading, because it has nothing at all to do with probability.

which we cannot necessarily predict the outcome of a particular choice, but we may know the range of possibilities. In reality, computations may depend on information that is out of the programmer's control, such as input from the user or actions of other processes in the system. Nondeterminism is useful in modeling such situations.

The importance of nondeterminism is not limited to logics of programs. Indeed, the most important open problem in the field of computational complexity theory, the *P=NP* problem, is formulated in terms of nondeterminism.

1.4 Program Verification

Dynamic Logic and other program logics are meant to be useful tools for facilitating the process of producing correct programs. One need only look at the miasma of buggy software to understand the dire need for such tools. But before we can produce correct software, we need to know what it means for it to be correct. It is not good enough to have some vague idea of what is supposed to happen when a program is run or to observe it running on some collection of inputs. In order to apply formal verification tools, we must have a formal specification of correctness for the verification tools to work with.

In general, a *correctness specification* is a formal description of how the program is supposed to behave. A given program is *correct* with respect to a correctness specification if its behavior fulfills that specification. For the gcd program of Example 1, the correctness might be specified informally by the assertion

> If the input values of x and y are positive integers c and d, respectively, then
>
> (i) the output value of x is the gcd of c and d, and
> (ii) the program halts.

Of course, in order to work with a formal verification system, these properties must be expressed formally in a language such as first-order logic.

The assertion (ii) is part of the correctness specification because programs do not necessarily halt, but may produce infinite traces for certain inputs. A finite trace, as for example the one produced by the gcd program above on input state (15,27,0), is called *halting*, *terminating*, or *convergent*. Infinite traces are called *looping* or *divergent*. For example, the program

$$\textbf{while } x > 7 \textbf{ do } x := x + 3$$

loops on input state $(8, \dots)$, producing the infinite trace

$$(8, \dots),\ (11, \dots),\ (14, \dots),\ \dots$$

Dynamic Logic can reason about the behavior of a program that is manifested in its input/output relation. It is not well suited to reasoning about program behavior manifested in intermediate states of a computation (although there are close relatives, such as Process Logic and Temporal Logic, that are). This is not to say that all interesting program behavior is captured by the input/output relation, and that other types of behavior are irrelevant or uninteresting. Indeed, the restriction to input/output relations is reasonable only when programs are supposed to halt after a finite time and yield output results. This approach will not be adequate for dealing with programs that normally are not supposed to halt, such as operating systems.

For programs that are supposed to halt, correctness criteria are traditionally given in the form of an *input/output specification* consisting of a formal relation between the input and output states that the program is supposed to maintain, along with a description of the set of input states on which the program is supposed to halt. The input/output relation of a program carries all the information necessary to determine whether the program is correct relative to such a specification. Dynamic Logic is well suited to this type of verification.

It is not always obvious what the correctness specification ought to be. Sometimes, producing a formal specification of correctness is as difficult as producing the program itself, since both must be written in a formal language. Moreover, specifications are as prone to bugs as programs. Why bother then? Why not just implement the program with some vague specification in mind?

There are several good reasons for taking the effort to produce formal specifications:

1. Often when implementing a large program from scratch, the programmer may have been given only a vague idea of what the finished product is supposed to do. This is especially true when producing software for a less technically inclined employer. There may be a rough informal description available, but the minor details are often left to the programmer. It is very often the case that a large part of the programming process consists of taking a vaguely specified problem and making it precise. The process of formulating the problem precisely can be considered a *definition* of what the program is supposed to do. And it is just good programming practice to have a very clear idea of what we want to do before we start doing it.

2. In the process of formulating the specification, several unforeseen cases may become apparent, for which it is not clear what the appropriate action of the program should be. This is especially true with error handling and other exceptional situations. Formulating a specification can define the action of the program in such situations and thereby tie up loose ends.

3. The process of formulating a rigorous specification can sometimes suggest ideas for implementation, because it forces us to isolate the issues that drive design decisions. When we know all the ways our data are going to be accessed, we are in a better position to choose the right data structures that optimize the tradeoffs between efficiency and generality.

4. The specification is often expressed in a language quite different from the programming language. The specification is *functional*—it tells *what* the program is supposed to do—as opposed to *imperative*—*how* to do it. It is often easier to specify the desired functionality independent of the details of how it will be implemented. For example, we can quite easily express what it means for a number x to be the gcd of y and z in first-order logic without even knowing how to compute it.

5. Verifying that a program meets its specification is a kind of sanity check. It allows us to give two solutions to the problem—once as a functional specification, and once as an algorithmic implementation—and lets us verify that the two are compatible. Any incompatibilities between the program and the specification are either bugs in the program, bugs in the specification, or both. The cycle of refining the specification, modifying the program to meet the specification, and reverifying until the process converges can lead to software in which we have much more confidence.

Partial and Total Correctness

Typically, a program is designed to implement some functionality. As mentioned above, that functionality can often be expressed formally in the form of an input/output specification. Concretely, such a specification consists of an *input condition* or *precondition* φ and an *output condition* or *postcondition* ψ. These are properties of the input state and the output state, respectively, expressed in some formal language such as the first-order language of the domain of computation. The program is supposed to halt in a state satisfying the output condition whenever the input state satisfies the input condition. We say that a program is *partially correct* with respect to a given input/output specification φ, ψ if, whenever the program is started in a state satisfying the input condition φ, then if and when it ever halts, it does so in a state satisfying the output condition ψ. The definition of partial correctness does not stipulate that the program halts; this is what we mean by *partial*.

A program is *totally correct* with respect to an input/output specification φ, ψ if

- it is partially correct with respect to that specification; and
- it halts whenever it is started in a state satisfying the input condition φ.

The input/output specification imposes no requirements when the input state does not satisfy the input condition φ—the program might as well loop infinitely or erase memory. This is the "garbage in, garbage out" philosophy. If we really do care what the program does on some of those input states, then we had better rewrite the input condition to include them and say formally what we want to happen in those cases.

For example, in the gcd program of Example 1, the output condition ψ might be the condition (i) stating that the output value of x is the gcd of the input values of x and y. We can express this completely formally in the language of first-order number theory. We may try to start off with the input specification $\varphi_0 = \mathbf{1}$ (*true*); that is, no restrictions on the input state at all. Unfortunately, if the initial value of y is 0 and x is negative, the final value of x will be the same as the initial value, thus negative. If we expect all gcds to be positive, this would be wrong. Another problematic situation arises when the initial values of x and y are both 0; in this case the gcd is not defined. Therefore, the program as written is not partially correct with respect to the specification φ_0, ψ.

We can remedy the situation by providing an input specification that rules out these troublesome input values. We can limit the input states to those in which x and y are both nonnegative and not both zero by taking the input specification

$$\varphi_1 \quad = \quad (x \geq 0 \wedge y > 0) \ \vee \ (x > 0 \wedge y \geq 0).$$

The gcd program of Example 1 above would be partially correct with respect to the specification φ_1, ψ. It is also totally correct, since the program halts on all inputs satisfying φ_1.

Perhaps we want to allow any input in which not both x and y are zero. In that case, we should use the input specification $\varphi_2 = \neg(x = 0 \wedge y = 0)$. But then the program of Example 1 is not partially correct with respect to φ_2, ψ; we must amend the program to produce the correct (positive) gcd on negative inputs.

1.5 Exogenous and Endogenous Logics

There are two main approaches to modal logics of programs: the *exogenous* approach, exemplified by Dynamic Logic and its precursor Hoare Logic ([Hoare, 1969]), and the *endogenous* approach, exemplified by Temporal Logic and its precursor, the invariant assertions method of [Floyd, 1967]. A logic is *exogenous* if its programs are explicit in the language. Syntactically, a Dynamic Logic program is a well-formed expression built inductively from primitive programs using a small set of program operators. Semantically, a program is interpreted as its input/output relation. The relation denoted by a compound program is determined by the relations denoted by its parts. This aspect of *compositionality* allows analysis by structural induction.

The importance of compositionality is discussed in [van Emde Boas, 1978]. In Temporal Logic, the program is fixed and is considered part of the structure over which the logic is interpreted. The current location in the program during execution is stored in a special variable for that purpose, called the *program counter*, and is part of the state along with the values of the program variables. Instead of program operators, there are temporal operators that describe how the program variables, including the program counter, change with time. Thus Temporal Logic sacrifices compositionality for a less restricted formalism. We discuss Temporal Logic further in Section 14.2.

2 PROPOSITIONAL DYNAMIC LOGIC (PDL)

Propositional Dynamic Logic (PDL) plays the same role in Dynamic Logic that classical propositional logic plays in classical predicate logic. It describes the properties of the interaction between programs and propositions that are independent of the domain of computation. Since PDL is a subsystem of first-order DL, we can be sure that all properties of PDL that we discuss in this section will also be valid in first-order DL.

Since there is no domain of computation in PDL, there can be no notion of assignment to a variable. Instead, primitive programs are interpreted as arbitrary binary relations on an abstract set of states K. Likewise, primitive assertions are just atomic propositions and are interpreted as arbitrary subsets of K. Other than this, no special structure is imposed.

This level of abstraction may at first appear too general to say anything of interest. On the contrary, it is a very natural level of abstraction at which many fundamental relationships between programs and propositions can be observed.

For example, consider the PDL formula

$$[\alpha](\varphi \wedge \psi) \quad \leftrightarrow \quad [\alpha]\varphi \wedge [\alpha]\psi. \tag{4}$$

The left-hand side asserts that the formula $\varphi \wedge \psi$ must hold after the execution of program α, and the right-hand side asserts that φ must hold after execution of α and so must ψ. The formula (4) asserts that these two statements are equivalent. This implies that to verify a conjunction of two postconditions, it suffices to verify each of them separately. The assertion (4) holds universally, regardless of the domain of computation and the nature of the particular α, φ, and ψ.

As another example, consider

$$[\alpha\,;\,\beta]\varphi \quad \leftrightarrow \quad [\alpha][\beta]\varphi. \tag{5}$$

The left-hand side asserts that after execution of the composite program $\alpha\,;\,\beta$, φ must hold. The right-hand side asserts that after execution of the program α, $[\beta]\varphi$ must hold, which in turn says that after execution of β, φ must hold. The formula (5) asserts the logical equivalence of these two statements. It holds regardless of the nature of α, β, and φ. Like (4), (5) can be used to simplify the verification of complicated programs.

As a final example, consider the assertion

$$[\alpha]p \quad \leftrightarrow \quad [\beta]p \tag{6}$$

where p is a primitive proposition symbol and α and β are programs. If this formula is true under all interpretations, then α and β are *equivalent* in the sense that they behave identically with respect to any property expressible in PDL or any formal system containing PDL as a subsystem. This is because the assertion will hold for any substitution instance of (6). For example, the two programs

$$\begin{array}{rcl} \alpha & = & \textbf{if } \varphi \textbf{ then } \gamma \textbf{ else } \delta \\ \beta & = & \textbf{if } \neg\varphi \textbf{ then } \delta \textbf{ else } \gamma \end{array}$$

are equivalent in the sense of (6).

2.1 Syntax

Syntactically, PDL is a blend of three classical ingredients: propositional logic, modal logic, and the algebra of regular expressions. There are several versions of PDL, depending on the choice of program operators allowed. In this section we will introduce the basic version, called *regular* PDL. Variations of this basic version will be considered in later sections.

The language of regular PDL has expressions of two sorts: *propositions* or *formulas* $\varphi, \psi, \ldots$ and *programs* $\alpha, \beta, \gamma, \ldots$. There are countably many *atomic symbols* of each sort. Atomic programs are denoted $a, b, c, \ldots$ and the set of all atomic programs is denoted Π_0. Atomic propositions are denoted $p, q, r, \ldots$ and the set of all atomic propositions is denoted Φ_0. The set of all programs is denoted Π and the set of all propositions is denoted Φ. Programs and propositions are built inductively from the atomic ones using the following operators:

Propositional operators:

$\rightarrow$	implication
$\mathbf{0}$	falsity

Program operators:

;	composition
$\cup$	choice
*	iteration

Mixed operators:

[ ]	necessity
?	test

The definition of programs and propositions is by mutual induction. All atomic programs are programs and all atomic propositions are propositions. If φ, ψ are propositions and α, β are programs, then

$\varphi \rightarrow \psi$	propositional implication
$\mathbf{0}$	propositional falsity
$[\alpha]\varphi$	program necessity

are propositions and

$\alpha\,;\,\beta$	sequential composition
$\alpha \cup \beta$	nondeterministic choice
α^*	iteration
$\varphi?$	test

are programs. In more formal terms, we define the set Π of all programs and the set Φ of all propositions to be the smallest sets such that

- $\Phi_0 \subseteq \Phi$
- $\Pi_0 \subseteq \Pi$
- if $\varphi, \psi \in \Phi$, then $\varphi \to \psi \in \Phi$ and $\mathbf{0} \in \Phi$
- if $\alpha, \beta \in \Pi$, then $\alpha;\beta$, $\alpha \cup \beta$, and $\alpha^* \in \Pi$
- if $\alpha \in \Pi$ and $\varphi \in \Phi$, then $[\alpha]\varphi \in \Phi$
- if $\varphi \in \Phi$ then $\varphi? \in \Pi$.

Note that the inductive definitions of programs Π and propositions Φ are intertwined and cannot be separated. The definition of propositions depends on the definition of programs because of the construct $[\alpha]\varphi$, and the definition of programs depends on the definition of propositions because of the construct $\varphi?$. Note also that we have allowed all formulas as tests. This is the *rich test* version of PDL.

Compound programs and propositions have the following intuitive meanings:

$[\alpha]\varphi$	"It is necessary that after executing α, φ is true."
$\alpha;\beta$	"Execute α, then execute β."
$\alpha \cup \beta$	"Choose either α or β nondeterministically and execute it."
α^*	"Execute α a nondeterministically chosen finite number of times (zero or more)."
$\varphi?$	"Test φ; proceed if true, fail if false."

We avoid parentheses by assigning precedence to the operators: unary operators, including $[\alpha]$, bind tighter than binary ones, and ; binds tighter than $\cup$. Thus the expression

$$[\alpha;\beta^* \cup \gamma^*]\varphi \vee \psi$$

should be read

$$([(\alpha;(\beta^*)) \cup (\gamma^*)]\varphi) \vee \psi.$$

Of course, parentheses can always be used to enforce a particular parse of an expression or to enhance readability. Also, under the semantics to be given in the next section, the operators ; and $\cup$ will turn out to be associative, so we may write $\alpha\,;\,\beta\,;\,\gamma$ and $\alpha \cup \beta \cup \gamma$ without ambiguity. We often omit the symbol ; and write the composition $\alpha\,;\,\beta$ as $\alpha\beta$.

The propositional operators $\wedge$, $\vee$, $\neg$, $\leftrightarrow$, and $\mathbf{1}$ can be defined from $\to$ and $\mathbf{0}$ in the usual way.

The possibility operator < > is the modal dual of the necessity operator [ ]. It is defined by

$$<\alpha>\varphi \quad \stackrel{\text{def}}{=} \quad \neg[\alpha]\neg\varphi.$$

The propositions $[\alpha]\varphi$ and $<\alpha>\varphi$ are read "box α φ" and "diamond α φ," respectively. The latter has the intuitive meaning, "There is a computation of α that terminates in a state satisfying φ."

One important difference between < > and [ ] is that $<\alpha>\varphi$ implies that α terminates, whereas $[\alpha]\varphi$ does not. Indeed, the formula $[\alpha]\mathbf{0}$ asserts that no computation of α terminates, and the formula $[\alpha]\mathbf{1}$ is always true, regardless of α.

In addition, we define

$$\begin{array}{rcl}
\mathbf{skip} & \stackrel{\text{def}}{=} & \mathbf{1}? \\
\mathbf{fail} & \stackrel{\text{def}}{=} & \mathbf{0}? \\
\mathbf{if}\ \varphi_1 \to \alpha_1 \ \mid \cdots \mid \ \varphi_n \to \alpha_n\ \mathbf{fi} & \stackrel{\text{def}}{=} & \varphi_1?;\alpha_1 \cup \cdots \cup \varphi_n?;\alpha_n \\
\mathbf{do}\ \varphi_1 \to \alpha_1 \ \mid \cdots \mid \ \varphi_n \to \alpha_n\ \mathbf{od} & \stackrel{\text{def}}{=} & (\bigcup_{i=1}^{n} \varphi_i?;\alpha_i)^*;(\bigwedge_{i=1}^{n} \neg\varphi_i)? \\
\mathbf{if}\ \varphi\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta & \stackrel{\text{def}}{=} & \mathbf{if}\ \varphi \to \alpha \ \mid\ \neg\varphi \to \beta\ \mathbf{fi} \\
& = & \varphi?;\alpha \cup \neg\varphi?;\beta \\
\mathbf{while}\ \varphi\ \mathbf{do}\ \alpha & \stackrel{\text{def}}{=} & \mathbf{do}\ \varphi \to \alpha\ \mathbf{od} \\
& = & (\varphi?;\alpha)^*;\neg\varphi? \\
\mathbf{repeat}\ \alpha\ \mathbf{until}\ \varphi & \stackrel{\text{def}}{=} & \alpha;\mathbf{while}\ \neg\varphi\ \mathbf{do}\ \alpha \\
& = & \alpha;(\neg\varphi?;\alpha)^*;\varphi? \\
\{\varphi\}\ \alpha\ \{\psi\} & \stackrel{\text{def}}{=} & \varphi \to [\alpha]\psi.
\end{array}$$

The programs **skip** and **fail** are the program that does nothing (no-op) and the failing program, respectively. The ternary **if-then-else** operator and the binary **while-do** operator are the usual *conditional* and *while loop* constructs found in conventional programming languages. The constructs **if-|-fi** and **do-|-od** are the *alternative guarded command* and *iterative guarded command* constructs, respectively. The construct $\{\varphi\}\ \alpha\ \{\psi\}$ is the Hoare partial correctness assertion. We will argue later that the formal definitions of these operators given above correctly model their intuitive behavior.

2.2 Semantics

The semantics of PDL comes from the semantics for modal logic. The structures over which programs and propositions of PDL are interpreted are called *Kripke frames* in honor of Saul Kripke, the inventor of the formal semantics of modal logic. A *Kripke frame* is a pair

$$\mathfrak{K} \quad = \quad (K,\ \mathfrak{m}_{\mathfrak{K}}),$$

where K is a set of elements $u, v, w, \ldots$ called *states* and $\mathfrak{m}_{\mathfrak{K}}$ is a *meaning function* assigning a subset of K to each atomic proposition and a binary relation on K to each atomic program. That is,

$$\begin{array}{rcll} \mathfrak{m}_{\mathfrak{K}}(p) & \subseteq & K, & p \in \Phi_0 \\ \mathfrak{m}_{\mathfrak{K}}(a) & \subseteq & K \times K, & a \in \Pi_0. \end{array}$$

We will extend the definition of the function $\mathfrak{m}_{\mathfrak{K}}$ by induction below to give a meaning to all elements of Π and Φ such that

$$\begin{array}{rcll} \mathfrak{m}_{\mathfrak{K}}(\varphi) & \subseteq & K, & \varphi \in \Phi \\ \mathfrak{m}_{\mathfrak{K}}(\alpha) & \subseteq & K \times K, & \alpha \in \Pi. \end{array}$$

Intuitively, we can think of the set $\mathfrak{m}_{\mathfrak{K}}(\varphi)$ as the set of states *satisfying* the proposition φ in the model $\mathfrak{K}$, and we can think of the binary relation $\mathfrak{m}_{\mathfrak{K}}(\alpha)$ as the set of input/output pairs of states of the program α.

Formally, the meanings $\mathfrak{m}_{\mathfrak{K}}(\varphi)$ of $\varphi \in \Phi$ and $\mathfrak{m}_{\mathfrak{K}}(\alpha)$ of $\alpha \in \Pi$ are defined by mutual induction on the structure of φ and α. The basis of the induction, which specifies the meanings of the atomic symbols $p \in \Phi_0$ and $a \in \Pi_0$, is already given in the specification of $\mathfrak{K}$. The meanings of compound propositions and programs are defined as follows.

$$\begin{array}{rcll}
\mathfrak{m}_{\mathfrak{K}}(\varphi \to \psi) & \stackrel{\mathrm{def}}{=} & (K - \mathfrak{m}_{\mathfrak{K}}(\varphi)) \cup \mathfrak{m}_{\mathfrak{K}}(\psi) & \\
\mathfrak{m}_{\mathfrak{K}}(\mathbf{0}) & \stackrel{\mathrm{def}}{=} & \varnothing & \\
\mathfrak{m}_{\mathfrak{K}}([\alpha]\varphi) & \stackrel{\mathrm{def}}{=} & K - (\mathfrak{m}_{\mathfrak{K}}(\alpha) \circ (K - \mathfrak{m}_{\mathfrak{K}}(\varphi))) & \\
 & = & \{u \mid \forall v \in K \text{ if } (u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ then } v \in \mathfrak{m}_{\mathfrak{K}}(\varphi)\} & \\
\mathfrak{m}_{\mathfrak{K}}(\alpha;\beta) & \stackrel{\mathrm{def}}{=} & \mathfrak{m}_{\mathfrak{K}}(\alpha) \circ \mathfrak{m}_{\mathfrak{K}}(\beta) & (7) \\
 & = & \{(u,v) \mid \exists w \in K\ (u,w) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ and } (w,v) \in \mathfrak{m}_{\mathfrak{K}}(\beta)\} & \\
\mathfrak{m}_{\mathfrak{K}}(\alpha \cup \beta) & \stackrel{\mathrm{def}}{=} & \mathfrak{m}_{\mathfrak{K}}(\alpha) \cup \mathfrak{m}_{\mathfrak{K}}(\beta) & \\
\mathfrak{m}_{\mathfrak{K}}(\alpha^*) & \stackrel{\mathrm{def}}{=} & \mathfrak{m}_{\mathfrak{K}}(\alpha)^* \ = \ \bigcup_{n \geq 0} \mathfrak{m}_{\mathfrak{K}}(\alpha)^n & (8) \\
\mathfrak{m}_{\mathfrak{K}}(\varphi?) & \stackrel{\mathrm{def}}{=} & \{(u,u) \mid u \in \mathfrak{m}_{\mathfrak{K}}(\varphi)\}. &
\end{array}$$

The operator $\circ$ in (7) is relational composition. In (8), the first occurrence of * is the iteration symbol of PDL, and the second is the reflexive transitive closure operator on binary relations. Thus (8) says that the program α^* is interpreted as the reflexive transitive closure of $\mathfrak{m}_{\mathfrak{K}}(\alpha)$.

We write $\mathfrak{K}, u \vDash \varphi$ and $u \in \mathfrak{m}_{\mathfrak{K}}(\varphi)$ interchangeably, and say that u *satisfies* φ in $\mathfrak{K}$, or that φ is *true* at state u in $\mathfrak{K}$. We may omit the $\mathfrak{K}$ and write $u \vDash \varphi$ when $\mathfrak{K}$ is understood. The notation $u \nvDash \varphi$ means that u does not satisfy φ, or in other words that $u \notin \mathfrak{m}_{\mathfrak{K}}(\varphi)$. In this notation, we can restate the definition above equivalently as follows:

$$\begin{aligned}
u \vDash \varphi \to \psi \quad &\overset{\text{def}}{\iff} \quad u \vDash \varphi \text{ implies } u \vDash \psi \\
u \nvDash \mathbf{0} \quad & \\
u \vDash [\alpha]\varphi \quad &\overset{\text{def}}{\iff} \quad \forall v \text{ if } (u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ then } v \vDash \varphi \\
(u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha\beta) \quad &\overset{\text{def}}{\iff} \quad \exists w\ (u,w) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ and } (w,v) \in \mathfrak{m}_{\mathfrak{K}}(\beta) \\
(u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha \cup \beta) \quad &\overset{\text{def}}{\iff} \quad (u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ or } (u,v) \in \mathfrak{m}_{\mathfrak{K}}(\beta) \\
(u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha^*) \quad &\overset{\text{def}}{\iff} \quad \exists n \geq 0\ \exists u_0, \ldots, u_n\ u = u_0,\ v = u_n, \\
&\qquad\quad \text{and } (u_i, u_{i+1}) \in \mathfrak{m}_{\mathfrak{K}}(\alpha),\ 0 \leq i \leq n-1 \\
(u,v) \in \mathfrak{m}_{\mathfrak{K}}(\varphi?) \quad &\overset{\text{def}}{\iff} \quad u = v \text{ and } u \vDash \varphi.
\end{aligned}$$

The defined operators inherit their meanings from these definitions:

$$\begin{aligned}
\mathfrak{m}_{\mathfrak{K}}(\varphi \vee \psi) &\overset{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\varphi) \cup \mathfrak{m}_{\mathfrak{K}}(\psi) \\
\mathfrak{m}_{\mathfrak{K}}(\varphi \wedge \psi) &\overset{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\varphi) \cap \mathfrak{m}_{\mathfrak{K}}(\psi) \\
\mathfrak{m}_{\mathfrak{K}}(\neg\varphi) &\overset{\text{def}}{=} K - \mathfrak{m}_{\mathfrak{K}}(\varphi) \\
\mathfrak{m}_{\mathfrak{K}}(<\alpha>\varphi) &\overset{\text{def}}{=} \{u \mid \exists v \in K\ (u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ and } v \in \mathfrak{m}_{\mathfrak{K}}(\varphi)\} \\
&= \mathfrak{m}_{\mathfrak{K}}(\alpha) \circ \mathfrak{m}_{\mathfrak{K}}(\varphi) \\
\mathfrak{m}_{\mathfrak{K}}(\mathbf{1}) &\overset{\text{def}}{=} K \\
\mathfrak{m}_{\mathfrak{K}}(\mathbf{skip}) &\overset{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\mathbf{1}?) \ = \ \iota, \text{ the identity relation} \\
\mathfrak{m}_{\mathfrak{K}}(\mathbf{fail}) &\overset{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\mathbf{0}?) \ = \ \varnothing.
\end{aligned}$$

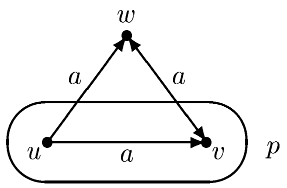
In addition, the **if-then-else**, **while-do**, and guarded commands inherit their semantics from the above definitions, and the input/output relations given by the formal semantics capture their intuitive operational meanings. For example, the relation associated with the program **while** φ **do** α is the set of pairs (u,v) for which there exist states $u_0, u_1, \ldots, u_n$, $n \geq 0$, such that $u = u_0$, $v = u_n$, $u_i \in \mathfrak{m}_{\mathfrak{K}}(\varphi)$ and $(u_i, u_{i+1}) \in \mathfrak{m}_{\mathfrak{K}}(\alpha)$ for $0 \leq i < n$, and $u_n \notin \mathfrak{m}_{\mathfrak{K}}(\varphi)$.

This version of PDL is usually called *regular* PDL and the elements of Π are called *regular programs* because of the primitive operators $\cup$, ;, and *, which are familiar from regular expressions. Programs can be viewed as regular expressions over the atomic programs and tests. In fact, it can be shown that if p is an atomic proposition symbol, then any two test-free programs α, β are equivalent as regular expressions—that is, they represent the same regular set—if and only if the formula $<\alpha>p \leftrightarrow <\beta>p$ is valid.

EXAMPLE 2. Let p be an atomic proposition, let a be an atomic program, and let $\mathfrak{K} = (K,\ \mathfrak{m}_{\mathfrak{K}})$ be a Kripke frame with

$$\begin{aligned} K &= \{u, v, w\} \\ \mathfrak{m}_{\mathfrak{K}}(p) &= \{u, v\} \\ \mathfrak{m}_{\mathfrak{K}}(a) &= \{(u, v), (u, w), (v, w), (w, v)\}. \end{aligned}$$

The following diagram illustrates $\mathfrak{K}$.



In this structure, $u \vDash <a>\neg p \wedge <a>p$, but $v \vDash [a]\neg p$ and $w \vDash [a]p$. Moreover, every state of $\mathfrak{K}$ satisfies the formula

$$<a^*>[(aa)^*]p \wedge <a^*>[(aa)^*]\neg p.$$

2.3 Computation Sequences

Let α be a program. Recall from Section 1.3 that a *finite computation sequence* of α is a finite-length string of atomic programs and tests representing a possible sequence of atomic steps that can occur in a halting execution of α. These strings are called *seqs* and are denoted $\sigma, \tau, \ldots$. The set of all such sequences is denoted $CS(\alpha)$. We use the word "possible" here loosely—$CS(\alpha)$ is determined by the syntax of α alone, and may contain strings that are never executed in any interpretation.

Formally, the set $CS(\alpha)$ is defined by induction on the structure of α:

$$\begin{aligned} CS(a) &\stackrel{\text{def}}{=} \{a\},\ a \text{ an atomic program} \\ CS(\varphi?) &\stackrel{\text{def}}{=} \{\varphi?\} \\ CS(\alpha;\beta) &\stackrel{\text{def}}{=} \{\gamma\delta \mid \gamma \in CS(\alpha),\ \delta \in CS(\beta)\} \\ CS(\alpha \cup \beta) &\stackrel{\text{def}}{=} CS(\alpha) \cup CS(\beta) \\ CS(\alpha^*) &\stackrel{\text{def}}{=} \bigcup_{n \geq 0} CS(\alpha^n) \end{aligned}$$

where $\alpha^0 = \mathbf{skip}$ and $\alpha^{n+1} = \alpha\alpha^n$. For example, if a is an atomic program and p is an atomic formula, then the program

$$\textbf{while } p \textbf{ do } a \quad = \quad (p?; a)^*; \neg p?$$

has as computation sequences all strings of the form

$$p?\ a\ p?\ a\ \cdots\ p?\ a\ \mathbf{skip}\ \neg p?.$$

Note that each finite computation sequence β of a program α is itself a program, and $CS(\beta) = \{\beta\}$. Moreover, the following proposition is not difficult to prove by induction on the structure of α:

PROPOSITION 3.

$$\mathfrak{m}_{\mathfrak{K}}(\alpha) \quad = \quad \bigcup_{\sigma \in CS(\alpha)} \mathfrak{m}_{\mathfrak{K}}(\sigma).$$

2.4 Satisfiability and Validity

The definitions of satisfiability and validity of propositions come from modal logic. Let $\mathfrak{K} = (K,\ \mathfrak{m}_{\mathfrak{K}})$ be a Kripke frame and let φ be a proposition. We have defined in Section 2.2 what it means for $\mathfrak{K}, u \vDash \varphi$. If $\mathfrak{K}, u \vDash \varphi$ for some $u \in K$, we say that φ is *satisfiable* in $\mathfrak{K}$. If φ is satisfiable in some $\mathfrak{K}$, we say that φ is *satisfiable*.

If $\mathfrak{K}, u \vDash \varphi$ for all $u \in K$, we write $\mathfrak{K} \vDash \varphi$ and say that φ is *valid* in $\mathfrak{K}$. If $\mathfrak{K} \vDash \varphi$ for all Kripke frames $\mathfrak{K}$, we write $\vDash \varphi$ and say that φ is *valid*.

If Σ is a set of propositions, we write $\mathfrak{K} \vDash \Sigma$ if $\mathfrak{K} \vDash \varphi$ for all $\varphi \in \Sigma$. A proposition ψ is said to be a *logical consequence* of Σ if $\mathfrak{K} \vDash \psi$ whenever $\mathfrak{K} \vDash \Sigma$, in which case we write $\Sigma \vDash \psi$. (Note that this is *not* the same as saying that $\mathfrak{K}, u \vDash \psi$ whenever $\mathfrak{K}, u \vDash \Sigma$.) We say that an inference rule

$$\frac{\varphi_1, \ldots, \varphi_n}{\varphi}$$

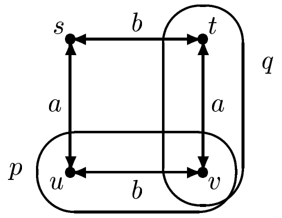
is *sound* if φ is a logical consequence of $\{\varphi_1, \ldots, \varphi_n\}$.

Satisfiability and validity are dual in the same sense that $\exists$ and $\forall$ are dual and `< >` and `[ ]` are dual: a proposition is valid (in $\mathfrak{K}$) if and only if its negation is not satisfiable (in $\mathfrak{K}$).

EXAMPLE 4. Let p, q be atomic propositions, let a, b be atomic programs, and let $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ be a Kripke frame with

$$\begin{array}{rcl} K & = & \{s, t, u, v\} \\ \mathfrak{m}_{\mathfrak{K}}(p) & = & \{u, v\} \\ \mathfrak{m}_{\mathfrak{K}}(q) & = & \{t, v\} \\ \mathfrak{m}_{\mathfrak{K}}(a) & = & \{(t, v), (v, t), (s, u), (u, s)\} \\ \mathfrak{m}_{\mathfrak{K}}(b) & = & \{(u, v), (v, u), (s, t), (t, s)\}. \end{array}$$

The following figure illustrates $\mathfrak{K}$.



The following formulas are valid in $\mathfrak{K}$.

$$p \leftrightarrow [(ab^*a)^*]p$$
$$q \leftrightarrow [(ba^*b)^*]q.$$

Also, let α be the program

$$\alpha = (aa \cup bb \cup (ab \cup ba)(aa \cup bb)^*(ab \cup ba))^*.$$

Thinking of α as a regular expression, α generates all words over the alphabet $\{a, b\}$ with an even number of occurrences of each of a and b. It can be shown that for any proposition φ, the proposition $\varphi \leftrightarrow [\alpha]\varphi$ is valid in $\mathfrak{K}$.

EXAMPLE 5. The formula

$$p \wedge [a^*]((p \to [a]\neg p) \wedge (\neg p \to [a]p)) \quad \leftrightarrow \quad [(aa)^*]p \wedge [a(aa)^*]\neg p$$

is valid. Both sides assert in different ways that p is alternately true and false along paths of execution of the atomic program a.

2.5 *Basic Properties*

THEOREM 6. *The following are valid formulas of* PDL*:*

(i) $<\alpha>(\varphi \vee \psi) \leftrightarrow <\alpha>\varphi \vee <\alpha>\psi$

(ii) $[\alpha](\varphi \wedge \psi) \leftrightarrow [\alpha]\varphi \wedge [\alpha]\psi$

(iii) $<\alpha>\varphi \wedge [\alpha]\psi \to <\alpha>(\varphi \wedge \psi)$

(iv) $[\alpha](\varphi \to \psi) \to ([\alpha]\varphi \to [\alpha]\psi)$

(v) $<\alpha>(\varphi \wedge \psi) \to <\alpha>\varphi \wedge <\alpha>\psi$

(vi) $[\alpha]\varphi \vee [\alpha]\psi \to [\alpha](\varphi \vee \psi)$

(vii) $<\alpha>\mathbf{0} \leftrightarrow \mathbf{0}$

(viii) $[\alpha]\varphi \leftrightarrow \neg<\alpha>\neg\varphi$.

(ix) $<\alpha \cup \beta>\varphi \leftrightarrow <\alpha>\varphi \vee <\beta>\varphi$

(x) $[\alpha \cup \beta]\varphi \;\leftrightarrow\; [\alpha]\varphi \wedge [\beta]\varphi$

(xi) $<\alpha\,;\,\beta>\varphi \;\leftrightarrow\; <\alpha><\beta>\varphi$

(xii) $[\alpha\,;\,\beta]\varphi \;\leftrightarrow\; [\alpha][\beta]\varphi$

(xiii) $<\varphi?>\psi \;\leftrightarrow\; (\varphi \wedge \psi)$

(xiv) $[\varphi?]\psi \;\leftrightarrow\; (\varphi \rightarrow \psi)$.

THEOREM 7. *The following are sound rules of inference of* PDL*:*

(i) *Modal generalization (GEN):*

$$\frac{\varphi}{[\alpha]\varphi}$$

(ii) *Monotonicity of* $<\alpha>$*:*

$$\frac{\varphi \rightarrow \psi}{<\alpha>\varphi \rightarrow <\alpha>\psi}$$

(iii) *Monotonicity of* $[\alpha]$*:*

$$\frac{\varphi \rightarrow \psi}{[\alpha]\varphi \rightarrow [\alpha]\psi}$$

The *converse operator* $^-$ is a program operator with semantics

$$\mathfrak{m}_{\mathfrak{K}}(\alpha^-) \;=\; \mathfrak{m}_{\mathfrak{K}}(\alpha)^- \;=\; \{(v,u) \mid (u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha)\}.$$

Intuitively, the converse operator allows us to "run a program backwards;" semantically, the input/output relation of the program α^- is the output/input relation of α. Although this is not always possible to realize in practice, it is nevertheless a useful expressive tool. For example, it gives us a convenient way to talk about *backtracking*, or rolling back a computation to a previous state.

THEOREM 8. *For any programs* α *and* β*,*

(i) $\mathfrak{m}_{\mathfrak{K}}((\alpha \cup \beta)^-) \;=\; \mathfrak{m}_{\mathfrak{K}}(\alpha^- \cup \beta^-)$

(ii) $\mathfrak{m}_{\mathfrak{K}}((\alpha\,;\,\beta)^-) \;=\; \mathfrak{m}_{\mathfrak{K}}(\beta^-\,;\,\alpha^-)$

(iii) $\mathfrak{m}_{\mathfrak{K}}(\varphi?^-) \;=\; \mathfrak{m}_{\mathfrak{K}}(\varphi?)$

(iv) $\mathfrak{m}_{\mathfrak{K}}(\alpha^{*-}) \;=\; \mathfrak{m}_{\mathfrak{K}}(\alpha^{-*})$

(v) $\mathfrak{m}_{\mathfrak{K}}(\alpha^{--}) \;=\; \mathfrak{m}_{\mathfrak{K}}(\alpha)$.

THEOREM 9. *The following are valid formulas of* PDL*:*

(i) $\varphi \rightarrow \mathtt{[}\alpha\mathtt{]}\mathtt{<}\alpha^{-}\mathtt{>}\varphi$

(ii) $\varphi \rightarrow \mathtt{[}\alpha^{-}\mathtt{]}\mathtt{<}\alpha\mathtt{>}\varphi$

(iii) $\mathtt{<}\alpha\mathtt{>}\mathtt{[}\alpha^{-}\mathtt{]}\varphi \rightarrow \varphi$

(iv) $\mathtt{<}\alpha^{-}\mathtt{>}\mathtt{[}\alpha\mathtt{]}\varphi \rightarrow \varphi$.

The iteration operator * is interpreted as the reflexive transitive closure operator on binary relations. It is the means by which iteration is coded in PDL. This operator differs from the other operators in that it is infinitary in nature, as reflected by its semantics:

$$\mathfrak{m}_{\mathfrak{K}}(\alpha^*) \quad = \quad \mathfrak{m}_{\mathfrak{K}}(\alpha)^* \quad = \quad \bigcup_{n<\omega} \mathfrak{m}_{\mathfrak{K}}(\alpha)^n$$

(see Section 2.2). This introduces a level of complexity to PDL beyond the other operators. Because of it, PDL is not compact: the set

$$\{\mathtt{<}\alpha^*\mathtt{>}\varphi\} \quad \cup \quad \{\neg\varphi,\ \neg\mathtt{<}\alpha\mathtt{>}\varphi,\ \neg\mathtt{<}\alpha^2\mathtt{>}\varphi,\ \ldots\} \tag{9}$$

is finitely satisfiable but not satisfiable. Because of this infinitary behavior, it is rather surprising that PDL should be decidable and that there should be a finitary complete axiomatization.

The properties of the * operator of PDL come directly from the properties of the reflexive transitive closure operator * on binary relations. In a nutshell, for any binary relation R, R^* is the $\subseteq$-least reflexive and transitive relation containing R.

THEOREM 10. *The following are valid formulas of* PDL*:*

(i) $\mathtt{[}\alpha^*\mathtt{]}\varphi \rightarrow \varphi$

(ii) $\varphi \rightarrow \mathtt{<}\alpha^*\mathtt{>}\varphi$

(iii) $\mathtt{[}\alpha^*\mathtt{]}\varphi \rightarrow \mathtt{[}\alpha\mathtt{]}\varphi$

(iv) $\mathtt{<}\alpha\mathtt{>}\varphi \rightarrow \mathtt{<}\alpha^*\mathtt{>}\varphi$

(v) $\mathtt{[}\alpha^*\mathtt{]}\varphi \leftrightarrow \mathtt{[}\alpha^*\alpha^*\mathtt{]}\varphi$

(vi) $\mathtt{<}\alpha^*\mathtt{>}\varphi \leftrightarrow \mathtt{<}\alpha^*\alpha^*\mathtt{>}\varphi$

(vii) $\mathtt{[}\alpha^*\mathtt{]}\varphi \leftrightarrow \mathtt{[}\alpha^{**}\mathtt{]}\varphi$

(viii) $\mathtt{<}\alpha^*\mathtt{>}\varphi \leftrightarrow \mathtt{<}\alpha^{**}\mathtt{>}\varphi$

(ix) $\mathtt{[}\alpha^*\mathtt{]}\varphi \leftrightarrow \varphi \wedge \mathtt{[}\alpha\mathtt{]}\mathtt{[}\alpha^*\mathtt{]}\varphi$.

(x) $<\alpha^*>\varphi \;\leftrightarrow\; \varphi \vee <\alpha><\alpha^*>\varphi$.

(xi) $[\alpha^*]\varphi \;\leftrightarrow\; \varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)$.

(xii) $<\alpha^*>\varphi \;\leftrightarrow\; \varphi \vee <\alpha^*>(\neg\varphi \wedge <\alpha>\varphi)$.

Semantically, α^* is a reflexive and transitive relation containing α, and Theorem 10 captures this. That α^* is reflexive is captured in (ii); that it is transitive is captured in (vi); and that it contains α is captured in (iv). These three properties are captured by the single property (x).

Reflexive Transitive Closure and Induction

To prove properties of iteration, it is not enough to know that α^* is a reflexive and transitive relation containing α. So is the universal relation $K \times K$, and that is not very interesting. We also need some way of capturing the idea that α^* is the *least* reflexive and transitive relation containing α. There are several equivalent ways this can be done:

(RTC) The *reflexive transitive closure rule*:

$$\frac{(\varphi \vee <\alpha>\psi) \to \psi}{<\alpha^*>\varphi \to \psi}$$

(LI) The *loop invariance rule*:

$$\frac{\psi \to [\alpha]\psi}{\psi \to [\alpha^*]\psi}$$

(IND) The *induction axiom* (box form):

$$\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi) \quad \to \quad [\alpha^*]\varphi$$

(IND) The *induction axiom* (diamond form):

$$<\alpha^*>\varphi \quad \to \quad \varphi \vee <\alpha^*>(\neg\varphi \wedge <\alpha>\varphi)$$

The rule (RTC) is called the *reflexive transitive closure rule*. Its importance is best described in terms of its relationship to the valid PDL formula of Theorem 10(x). Observe that the right-to-left implication of this formula is obtained by substituting $<\alpha^*>\varphi$ for R in the expression

$$\varphi \vee <\alpha>R \quad \to \quad R. \tag{10}$$

Theorem 10(x) implies that $<\alpha^*>\varphi$ is a solution of (10); that is, (10) is valid when $<\alpha^*>\varphi$ is substituted for R. The rule (RTC) says that $<\alpha^*>\varphi$ is the *least* such solution with respect to logical implication. That is, it is the least PDL-definable set of states that when substituted for R in (10) results in a valid formula.

The dual propositions labeled (IND) are jointly called the PDL *induction axiom*. Intuitively, the box form of (IND) says, "If φ is true initially, and if, after any number of iterations of the program α, the truth of φ is preserved by one more iteration of α, then φ will be true after any number of iterations of α." The diamond form of (IND) says, "If it is possible to reach a state satisfying φ in some number of iterations of α, then either φ is true now, or it is possible to reach a state in which φ is false but becomes true after one more iteration of α."

Note that the box form of (IND) bears a strong resemblance to the induction axiom of Peano arithmetic:

$$\varphi(0) \land \forall n\ (\varphi(n) \to \varphi(n+1)) \quad \to \quad \forall n\ \varphi(n).$$

Here $\varphi(0)$ is the basis of the induction and $\forall n\ (\varphi(n) \to \varphi(n+1))$ is the induction step, from which the conclusion $\forall n\ \varphi(n)$ can be drawn. In the PDL axiom (IND), the basis is φ and the induction step is $[\alpha^*](\varphi \to [\alpha]\varphi)$, from which the conclusion $[\alpha^*]\varphi$ can be drawn.

2.6 Encoding Hoare Logic

The Hoare partial correctness assertion $\{\varphi\}\ \alpha\ \{\psi\}$ is encoded as $\varphi \to [\alpha]\psi$ in PDL. The following theorem says that under this encoding, Dynamic Logic subsumes Hoare Logic.

THEOREM 11. *The following rules of Hoare Logic are derivable in* PDL*:*

(i) *Composition rule:*

$$\frac{\{\varphi\}\,\alpha\,\{\sigma\}, \quad \{\sigma\}\,\beta\,\{\psi\}}{\{\varphi\}\,\alpha\,;\,\beta\,\{\psi\}}$$

(ii) *Conditional rule:*

$$\frac{\{\varphi \land \sigma\}\,\alpha\,\{\psi\}, \quad \{\neg\varphi \land \sigma\}\,\beta\,\{\psi\}}{\{\sigma\}\ \textbf{if}\ \varphi\ \textbf{then}\ \alpha\ \textbf{else}\ \beta\ \{\psi\}}$$

(iii) ***While*** *rule:*

$$\frac{\{\varphi \land \psi\}\,\alpha\,\{\psi\}}{\{\psi\}\ \textbf{while}\ \varphi\ \textbf{do}\ \alpha\ \{\neg\varphi \land \psi\}}$$

(iv) *Weakening rule:*

$$\frac{\varphi' \to \varphi, \quad \{\varphi\}\,\alpha\,\{\psi\}, \quad \psi \to \psi'}{\{\varphi'\}\,\alpha\,\{\psi'\}}$$

3 FILTRATION AND DECIDABILITY

The *small model property* for PDL says that if φ is satisfiable, then it is satisfied at a state in a Kripke frame with no more than $2^{|\varphi|}$ states, where $|\varphi|$ is the number of symbols of φ. This result and the technique used to prove it, called *filtration*, come directly from modal logic. This immediately gives a naive decision procedure for the satisfiability problem for PDL: to determine whether φ is satisfiable, construct all Kripke frames with at most $2^{|\varphi|}$ states and check whether φ is satisfied at some state in one of them. Considering only interpretations of the primitive formulas and primitive programs appearing in φ, there are roughly $2^{2^{|\varphi|}}$ such models, so this algorithm is too inefficient to be practical. A more efficient algorithm will be described in Section 5.

3.1 The Fischer–Ladner Closure

Many proofs in simpler modal systems use induction on the well-founded subformula relation. In PDL, the situation is complicated by the simultaneous inductive definitions of programs and propositions and by the behavior of the * operator, which make the induction proofs somewhat tricky. Nevertheless, we can still use the well-founded subexpression relation in inductive proofs. Here an *expression* can be either a program or a proposition. Either one can be a subexpression of the other because of the mixed operators [ ] and ?.

We start by defining two functions

$$\begin{array}{rcl} FL & : & \Phi \;\rightarrow\; 2^{\Phi} \\ FL^{\Box} & : & \{[\alpha]\varphi \mid \alpha \in \Psi,\ \varphi \in \Phi\} \;\rightarrow\; 2^{\Phi} \end{array}$$

by simultaneous induction. The set $FL(\varphi)$ is called the *Fischer–Ladner closure* of φ. The filtration construction for PDL uses the Fischer–Ladner closure of a given formula where the corresponding proof for propositional modal logic would use the set of subformulas.

The functions FL and $FL^{\Box}$ are defined inductively as follows:

(a) $FL(p) \stackrel{\text{def}}{=} \{p\}$, p an atomic proposition

(b) $FL(\varphi \rightarrow \psi) \stackrel{\text{def}}{=} \{\varphi \rightarrow \psi\} \cup FL(\varphi) \cup FL(\psi)$

(c) $FL(\mathbf{0}) \stackrel{\text{def}}{=} \{\mathbf{0}\}$

(d) $FL([\alpha]\varphi) \stackrel{\text{def}}{=} FL^{\Box}([\alpha]\varphi) \cup FL(\varphi)$

(e) $FL^{\Box}([a]\varphi) \stackrel{\text{def}}{=} \{[a]\varphi\}$, a an atomic program

(f) $FL^{\Box}([\alpha \cup \beta]\varphi) \stackrel{\text{def}}{=} \{[\alpha \cup \beta]\varphi\} \cup FL^{\Box}([\alpha]\varphi) \cup FL^{\Box}([\beta]\varphi)$

(g) $FL^{\Box}([\alpha\,;\,\beta]\varphi) \stackrel{\mathrm{def}}{=} \{[\alpha\,;\,\beta]\varphi\} \cup FL^{\Box}([\alpha]\,[\beta]\varphi) \cup FL^{\Box}([\beta]\varphi)$

(h) $FL^{\Box}([\alpha^*]\varphi) \stackrel{\mathrm{def}}{=} \{[\alpha^*]\varphi\} \cup FL^{\Box}([\alpha]\,[\alpha^*]\varphi)$

(i) $FL^{\Box}([\psi?]\varphi) \stackrel{\mathrm{def}}{=} \{[\psi?]\varphi\} \cup FL(\psi)$.

This definition is apparently quite a bit more involved than for mere subexpressions. In fact, at first glance it may appear circular because of the rule (h). The auxiliary function $FL^{\Box}$ is introduced for the express purpose of avoiding any such circularity. It is defined only for formulas of the form $[\alpha]\varphi$ and intuitively produces those elements of $FL([\alpha]\varphi)$ obtained by breaking down α and ignoring φ.

LEMMA 12.

(i) *If* $[\alpha]\psi \in FL(\varphi)$, *then* $\psi \in FL(\varphi)$.

(ii) *If* $[\rho?]\psi \in FL(\varphi)$, *then* $\rho \in FL(\varphi)$.

(iii) *If* $[\alpha \cup \beta]\psi \in FL(\varphi)$, *then* $[\alpha]\psi \in FL(\varphi)$ *and* $[\beta]\psi \in FL(\varphi)$.

(iv) *If* $[\alpha\,;\,\beta]\psi \in FL(\varphi)$, *then* $[\alpha]\,[\beta]\psi \in FL(\varphi)$ *and* $[\beta]\psi \in FL(\varphi)$.

(v) *If* $[\alpha^*]\psi \in FL(\varphi)$, *then* $[\alpha]\,[\alpha^*]\psi \in FL(\varphi)$.

Even after convincing ourselves that the definition is noncircular, it may not be clear how the size of $FL(\varphi)$ depends on the length of φ. Indeed, the right-hand side of rule (h) involves a formula that is larger than the formula on the left-hand side. However, it can be shown by induction on subformulas that the relationship is linear:

LEMMA 13.

(i) *For any formula* φ, $\#FL(\varphi) \leq |\varphi|$.

(ii) *For any formula* $[\alpha]\varphi$, $\#FL^{\Box}([\alpha]\varphi) \leq |\alpha|$.

3.2 Filtration

Given a PDL proposition φ and a Kripke frame $\mathfrak{K} = (K,\ \mathfrak{m}_{\mathfrak{K}})$, we define a new frame $\mathfrak{K}/FL(\varphi) = (K/FL(\varphi),\ \mathfrak{m}_{\mathfrak{K}/FL(\varphi)})$, called the *filtration of* $\mathfrak{K}$ *by* $FL(\varphi)$, as follows. Define a binary relation $\equiv$ on states of $\mathfrak{K}$ by:

$$u \equiv v \quad \stackrel{\mathrm{def}}{\iff} \quad \forall\psi \in FL(\varphi)\ (u \in \mathfrak{m}_{\mathfrak{K}}(\psi) \Leftrightarrow v \in \mathfrak{m}_{\mathfrak{K}}(\psi)).$$

In other words, we collapse states u and v if they are not distinguishable by any formula of $FL(\varphi)$. Let

$$\begin{aligned}
[u] &\stackrel{\text{def}}{=} \{v \mid v \equiv u\} \\
K/FL(\varphi) &\stackrel{\text{def}}{=} \{[u] \mid u \in K\} \\
\mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(p) &\stackrel{\text{def}}{=} \{[u] \mid u \in \mathfrak{m}_{\mathfrak{K}}(p)\}, \quad p \text{ an atomic proposition} \\
\mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(a) &\stackrel{\text{def}}{=} \{([u],[v]) \mid (u,v) \in \mathfrak{m}_{\mathfrak{K}}(a)\}, \quad a \text{ an atomic program.}
\end{aligned}$$

The map $\mathfrak{m}_{\mathfrak{K}/FL(\varphi)}$ is extended inductively to compound propositions and programs as described in Section 2.2.

The following key lemma relates $\mathfrak{K}$ and $\mathfrak{K}/FL(\varphi)$. Most of the difficulty in the following lemma is in the correct formulation of the induction hypotheses in the statement of the lemma. Once this is done, the proof is a fairly straightforward induction on the well-founded subexpression relation.

LEMMA 14 (Filtration Lemma). *Let $\mathfrak{K}$ be a Kripke frame and let u, v be states of $\mathfrak{K}$.*

(i) *For all $\psi \in FL(\varphi)$, $u \in \mathfrak{m}_{\mathfrak{K}}(\psi)$ iff $[u] \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\psi)$.*

(ii) *For all $[\alpha]\psi \in FL(\varphi)$,*

 (a) *if $(u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha)$ then $([u],[v]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\alpha)$;*

 (b) *if $([u],[v]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\alpha)$ and $u \in \mathfrak{m}_{\mathfrak{K}}([\alpha]\psi)$, then $v \in \mathfrak{m}_{\mathfrak{K}}(\psi)$.*

Using the filtration lemma, we can prove the small model theorem easily.

THEOREM 15 (Small Model Theorem). *Let φ be a satisfiable formula of* PDL. *Then φ is satisfied in a Kripke frame with no more than $2^{|\varphi|}$ states.*

Proof. If φ is satisfiable, then there is a Kripke frame $\mathfrak{K}$ and state $u \in \mathfrak{K}$ with $u \in \mathfrak{m}_{\mathfrak{K}}(\varphi)$. Let $FL(\varphi)$ be the Fischer-Ladner closure of φ. By the filtration lemma (Lemma 14), $[u] \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\varphi)$. Moreover, $\mathfrak{K}/FL(\varphi)$ has no more states than the number of truth assignments to formulas in $FL(\varphi)$, which by Lemma 13(i) is at most $2^{|\varphi|}$. ■

It follows immediately that the satisfiability problem for PDL is decidable, since there are only finitely many possible Kripke frames of size at most $2^{|\varphi|}$ to check, and there is a polynomial-time algorithm to check whether a given formula is satisfied at a given state in a given Kripke frame. A more efficient algorithm exists (see Section 5).

The completeness proof for PDL also makes use of the filtration lemma (Lemma 14), but in a somewhat stronger form. We need to know that it also holds for *nonstandard Kripke frames* as well as the standard Kripke frames defined in Section 2.2.

A *nonstandard Kripke frame* is any structure $\mathfrak{N} = (N, \mathfrak{m}_{\mathfrak{N}})$ that is a Kripke frame in the sense of Section 2.2 in every respect, except that $\mathfrak{m}_{\mathfrak{N}}(\alpha^*)$ need not be the reflexive transitive closure of $\mathfrak{m}_{\mathfrak{N}}(\alpha)$, but only a reflexive, transitive binary relation containing $\mathfrak{m}_{\mathfrak{N}}(\alpha)$ satisfying the PDL axioms for * (Axioms 17(vii) and (viii) of Section 4.1).

LEMMA 16 (Filtration for Nonstandard Models). *Let $\mathfrak{N}$ be a nonstandard Kripke frame and let u, v be states of $\mathfrak{N}$.*

(i) *For all $\psi \in FL(\varphi)$, $u \in \mathfrak{m}_{\mathfrak{N}}(\psi)$ iff $[u] \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\psi)$.*

(ii) *For all $[\alpha]\psi \in FL(\varphi)$,*

  (a) *if $(u, v) \in \mathfrak{m}_{\mathfrak{N}}(\alpha)$ then $([u], [v]) \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\alpha)$;*

  (b) *if $([u], [v]) \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\alpha)$ and $u \in \mathfrak{m}_{\mathfrak{N}}([\alpha]\psi)$, then $v \in \mathfrak{m}_{\mathfrak{N}}(\psi)$.*

4 DEDUCTIVE COMPLETENESS OF PDL

4.1 A Deductive System

The following list of axioms and rules constitutes a sound and complete Hilbert-style deductive system for PDL.

Axiom System 17.

(i) *Axioms for propositional logic*

(ii) $[\alpha](\varphi \to \psi) \quad \to \quad ([\alpha]\varphi \to [\alpha]\psi)$

(iii) $[\alpha](\varphi \wedge \psi) \quad \leftrightarrow \quad [\alpha]\varphi \wedge [\alpha]\psi$

(iv) $[\alpha \cup \beta]\varphi \quad \leftrightarrow \quad [\alpha]\varphi \wedge [\beta]\varphi$

(v) $[\alpha\, ;\, \beta]\varphi \quad \leftrightarrow \quad [\alpha][\beta]\varphi$

(vi) $[\psi?]\varphi \quad \leftrightarrow \quad (\psi \to \varphi)$

(vii) $\varphi \wedge [\alpha][\alpha^*]\varphi \quad \leftrightarrow \quad [\alpha^*]\varphi$

(viii) $\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi) \quad \to \quad [\alpha^*]\varphi$

In PDL *with converse* $^-$*, we also include*

(ix) $\varphi \quad \to \quad [\alpha]\langle\alpha^-\rangle\varphi$

(x) $\varphi \quad \to \quad [\alpha^-]\langle\alpha\rangle\varphi$

Rules of Inference

(MP) $\dfrac{\varphi, \quad \varphi \to \psi}{\psi}$

(GEN) $\dfrac{\varphi}{[\alpha]\varphi}$

□

The axioms (ii) and (iii) and the two rules of inference are not particular to PDL, but come from modal logic. The rules (MP) and (GEN) are called *modus ponens* and *(modal) generalization*, respectively.

Axiom (viii) is called the PDL *induction axiom*. Intuitively, (viii) says: "Suppose φ is true in the current state, and suppose that after any number of iterations of α, if φ is still true, then it will be true after one more iteration of α. Then φ will be true after any number of iterations of α." In other words, if φ is true initially, and if the truth of φ is preserved by the program α, then φ will be true after any number of iterations of α.

We write $\vdash \varphi$ if the proposition φ is a theorem of this system, and say that φ is *consistent* if $\nvdash \neg\varphi$; that is, if it is not the case that $\vdash \neg\varphi$. A set Σ of propositions is *consistent* if all finite conjunctions of elements of Σ are consistent.

The soundness of these axioms and rules over Kripke frames can be established by elementary arguments in relational algebra using the semantics of Section 2.2.

We write $\vdash \varphi$ if the formula φ is provable in this deductive system. A formula φ is *consistent* if $\nvdash \neg\varphi$, that is, if it is not the case that $\vdash \neg\varphi$; that a finite set Σ of formulas is *consistent* if its conjunction $\bigwedge \Sigma$ is consistent; and that an infinite set of formulas is *consistent* if every finite subset is consistent.

Axiom System 17 is complete: all valid formulas of PDL are theorems. This fact can be proved by constructing a nonstandard Kripke frame from maximal consistent sets of formulas, then using the filtration lemma for nonstandard models (Lemma 16) to collapse this nonstandard model to a finite standard model.

THEOREM 18 (Completeness of PDL). *If* $\vDash \varphi$ *then* $\vdash \varphi$.

In classical logics, a completeness theorem of the form of Theorem 18 can be adapted to handle the relation of logical consequence $\varphi \models \psi$ between formulas because of the deduction theorem, which says

$$\varphi \vdash \psi \quad \Leftrightarrow \quad \vdash \varphi \rightarrow \psi.$$

Unfortunately, the deduction theorem fails in PDL, as can be seen by taking $\psi = [a]p$ and $\varphi = p$. However, the following result allows Theorem 18, as well as the deterministic exponential-time satisfiability algorithm described in the next section, to be extended to handle the logical consequence relation:

THEOREM 19. *Let* φ *and* ψ *be any* PDL *formulas. Then*

$$\varphi \models \psi \quad \Leftrightarrow \quad \models [(a_1 \cup \cdots \cup a_n)^*]\varphi \rightarrow \psi,$$

where $a_1, \ldots, a_n$ *are all atomic programs appearing in* φ *or* ψ. *Allowing infinitary conjunctions, if* Σ *is a set of formulas in which only finitely many atomic programs appear, then*

$$\Sigma \models \psi \quad \Leftrightarrow \quad \models \bigwedge\{[(a_1 \cup \cdots \cup a_n)^*]\varphi \mid \varphi \in \Sigma\} \rightarrow \psi,$$

where $a_1, \ldots, a_n$ *are all atomic programs appearing in* Σ *or* ψ.

5 COMPLEXITY OF PDL

The small model theorem (Theorem 15) gives a naive deterministic algorithm for the satisfiability problem: construct all Kripke frames of at most $2^{|\varphi|}$ states and check whether φ is satisfied at any state in any of them. Although checking whether a given formula is satisfied in a given state of a given Kripke frame can be done quite efficiently, the naive satisfiability algorithm is highly inefficient. For one thing, the models constructed are of exponential size in the length of the given formula; for another, there are $2^{2^{O(|\varphi|)}}$ of them. Thus the naive satisfiability algorithm takes double exponential time in the worst case.

There is a more efficient algorithm [Pratt, 1979b] that runs in deterministic single-exponential time. One cannot expect to improve this significantly due to a corresponding lower bound.

THEOREM 20. *There is an exponential-time algorithm for deciding whether a given formula of* PDL *is satisfiable.*

THEOREM 21. *The satisfiability problem for* PDL *is EXPTIME-complete.*

COROLLARY 22. *There is a constant $c > 1$ such that the satisfiability problem for* PDL *is not solvable in deterministic time $c^{n/\log n}$, where n is the size of the input formula.*

EXPTIME-hardness can be established by constructing a formula of PDL whose models encode the computation of a given linear-space-bounded one-tape alternating Turing machine M on a given input x of length n over M's input alphabet. Since the membership problem for alternating polynomial-space machines is *EXPTIME*-hard ([Chandra *et al.*, 1981]), so is the satisfiability problem for PDL.

It is interesting to compare the complexity of satisfiability in PDL with the complexity of satisfiability in propositional logic. In the latter, satisfiability is *NP*-complete; but at present it is not known whether the two complexity classes *EXPTIME* and *NP* differ. Thus, as far as current knowledge goes, the satisfiability problem is no easier in the worst case for propositional logic than for its far richer superset PDL.

As we have seen, current knowledge does not permit a significant difference to be observed between the complexity of satisfiability in propositional logic and in PDL. However, there is one easily verified and important behavioral difference: propositional logic is *compact*, whereas PDL is not.

Compactness has significant implications regarding the relation of logical consequence. If a propositional formula φ is a consequence of a set Γ of propositional formulas, then it is already a consequence of some finite subset of Γ; but this is not true in PDL.

Recall that we write $\Gamma \vDash \varphi$ and say that φ is a *logical consequence* of Γ if φ satisfied in any state of any Kripke frame $\mathfrak{K}$ all of whose states satisfy all the formulas of Γ. That is, if $\mathfrak{K} \vDash \Gamma$, then $\mathfrak{K} \vDash \varphi$.

An alternative intepretation of logical consequence, not equivalent to the above, is that in any Kripke frame, the formula φ holds in any state satisfying all formulas in Γ. Allowing infinite conjunctions, we might write this as $\models \bigwedge \Gamma \rightarrow \varphi$. This is not the same as $\Gamma \models \varphi$, since $\models \bigwedge \Gamma \rightarrow \varphi$ implies $\Gamma \models \varphi$, but not necessarily vice versa. A counterexample is provided by $\Gamma = \{p\}$ and $\varphi = [a]p$. However, if Γ contains only finitely many atomic programs, we can reduce the problem $\Gamma \models \varphi$ to the problem $\models \bigwedge \Gamma' \rightarrow \varphi$ for a related Γ', as shown in Theorem 19.

Under either interpretation, compactness fails:

THEOREM 23. *There is an infinite set of formulas Γ and a formula φ such that $\models \bigwedge \Gamma \rightarrow \varphi$ (hence $\Gamma \models \varphi$), but for no proper subset $\Gamma' \subseteq \Gamma$ is it the case that $\Gamma' \models \varphi$ (hence neither is it the case that $\models \bigwedge \Gamma' \rightarrow \varphi$).*

As shown in Theorem 19, logical consequences $\Gamma \models \varphi$ for finite Γ are no more difficult to decide than validity of single formulas. But what if Γ is infinite? Here compactness is the key factor. If Γ is an r.e. set and the logic is compact, then the consequence problem is r.e.: to check whether $\Gamma \models \varphi$, the finite subsets of Γ can be effectively enumerated, and checking $\Gamma \models \varphi$ for finite Γ is a decidable problem.

Since compactness fails in PDL, this observation does us no good, even when Γ is known to be recursively enumerable. However, the following result shows that the situation is much worse than we might expect: even if Γ is taken to be the set of substitution instances of a single formula of PDL, the consequence problem becomes very highly undecidable. This is a rather striking manifestation of PDL's lack of compactness.

Let φ be a given formula. The set S_φ of *substitution instances* of φ is the set of all formulas obtained by substituting a formula for each atomic proposition appearing in φ.

THEOREM 24. *The problem of deciding whether $S_\varphi \models \psi$ is Π^1_1-complete. The problem is Π^1_1-hard even for a particular fixed φ.*

6 NONREGULAR PDL

In this section we enrich the class of regular programs in PDL by introducing programs whose control structure requires more than a finite automaton. For example, the class of *context-free programs* requires a pushdown automaton (PDA), and moving up from regular to context-free programs is really going from iterative programs to ones with parameterless recursive procedures. Several questions arise when enriching the class of programs of PDL, such as whether the expressive power of the logic grows, and if so whether the resulting logics are still decidable. It turns out that *any* nonregular program increases PDL's expressive power and that the validity problem for PDL with context-free programs is undecidable. The bulk of the section is then devoted to the difficult problem of trying to characterize the borderline between decidable and undecidable extensions. On the one hand, validity for PDL with the addition of even a single extremely simple nonregular program is already Π^1_1-complete; but on the other hand, when we add another equally simple program, the problem remains decidable. Besides these results, which pertain to very specific extensions, we discuss some broad decidability results that cover many languages, including some that are not even context-free. Since no similarly general undecidability results are known, we also address the weaker issue of whether nonregular extensions admit the finite model property and present a negative result that covers many cases.

6.1 Nonregular Programs

Consider the following self-explanatory program:

$$\textbf{while } p \textbf{ do } a\ ;\ \textbf{now do } b \textbf{ the same number of times} \tag{11}$$

This program is meant to represent the following set of computation sequences:

$$\{(p?\ ;\ a)^i\ ;\ \neg p?\ ;\ b^i \mid i \geq 0\}.$$

Viewed as a language over the alphabet $\{a, b, p, \neg p\}$, this set is not regular, thus cannot be programmed in PDL. However, it can be represented by the following parameterless recursive procedure:

proc V $\{$
  if p **then** $\{$ a ; **call** V ; b $\}$
    else return
  $\}$

The set of computation sequences of this program is captured by the context-free grammar

$$V \quad \to \quad \neg p? \mid p?aVb.$$

We are thus led to the idea of allowing context-free programs inside the boxes and diamonds of PDL. From a pragmatic point of view, this amounts to extending the logic with the ability to reason about parameterless recursive procedures. The particular representation of the context-free programs is unimportant; we can use pushdown automata, context-free grammars, recursive procedures, or any other formalism that can be effectively translated into these.

In the rest of this section, a number of specific programs will be of interest, and we employ special abbreviations for them. For example, we define:

$$\begin{aligned} a^\Delta b a^\Delta &\stackrel{\mathrm{def}}{=} \{a^i b a^i \mid i \geq 0\} \\ a^\Delta b^\Delta &\stackrel{\mathrm{def}}{=} \{a^i b^i \mid i \geq 0\} \\ b^\Delta a^\Delta &\stackrel{\mathrm{def}}{=} \{b^i a^i \mid i \geq 0\}. \end{aligned}$$

Note that $a^\Delta b^\Delta$ is really just a nondeterministic version of the program (11) in which there is simply no p to control the iteration. In fact, (11) could have been written in this notation as $(p?a)^\Delta \neg p? b^\Delta$.[2] In programming terms, we can compare the regular program $(ab)^*$ with the nonregular one $a^\Delta b^\Delta$ by observing that if a is "purchase a loaf of bread" and b is "pay \$1.00," then the former program captures the process of paying for each loaf when purchased, while the latter one captures the process of paying for them all at the end of the month.

It turns out that enriching PDL with even a single arbitrary nonregular program increases expressive power.

If L is any language over atomic programs and tests, then PDL + L is defined exactly as PDL, but with the additional syntax rule stating that for any formula φ, the expression $<L>\varphi$ is a new formula. The semantics of PDL + L is like that of PDL with the addition of the clause

$$\mathfrak{m}_{\mathfrak{K}}(L) \stackrel{\mathrm{def}}{=} \bigcup_{\beta \in L} \mathfrak{m}_{\mathfrak{K}}(\beta).$$

Note that PDL + L does not allow L to be used as a formation rule for new programs or to be combined with other programs. It is added to the programming language as a single new stand-alone program only.

If PDL_1 and PDL_2 are two extensions of PDL, we say that PDL_1 is *as expressive as* PDL_2 if for each formula φ of PDL_2 there is a formula ψ of PDL_1 such that $\vDash \varphi \leftrightarrow \psi$. If PDL_1 is as expressive as PDL_2 but PDL_2 is not as expressive as PDL_1, we say that PDL_1 is *strictly more expressive than* PDL_2.

Thus, one version of PDL is strictly more expressive than another if anything the latter can express the former can too, but there is something the former can express that the latter cannot.

[2]It is noteworthy that the results of this section do not depend on nondeterminism. For example, the negative Theorem 28 holds for the deterministic version (11) too. Also, most of the results in this section involve nonregular programs over atomic programs only, but can be generalized to allow tests as well.

A language is *test-free* if it is a subset of Π_0^*; that is, if its seqs contain no tests.

THEOREM 25. *If L is any nonregular test-free language, then* $\mathsf{PDL}+L$ *is strictly more expressive than* PDL.

We can view the decidability of regular PDL as showing that propositional-level reasoning about iterative programs is computable. We now wish to know if the same is true for recursive procedures. We define *context-free* PDL to be PDL extended with context-free programs, where a *context-free program* is one whose seqs form a context-free language. The precise syntax is unimportant, but for definiteness we might take as programs the set of context-free grammars G over atomic programs and tests and define

$$\mathfrak{m}_{\mathfrak{K}}(G) \quad \stackrel{\mathrm{def}}{=} \quad \bigcup_{\beta \in CS(G)} \mathfrak{m}_{\mathfrak{K}}(\beta),$$

where $CS(G)$ is the set of computation sequences generated by G as described in Section 1.3.

THEOREM 26. *The validity problem for context-free* PDL *is undecidable.*

Theorem 26 leaves several interesting questions unanswered. What is the level of undecidability of context-free PDL? What happens if we want to add only a small number of specific nonregular programs? The first of these questions arises from the fact that the equivalence problem for context-free languages is co-r.e.-complete, or complete for Π_1^0 in the arithmetic hierarchy. Hence, all Theorem 26 shows is that the validity problem for context-free PDL is Π_1^0-hard, while it might in fact be worse. The second question is far more general. We might be interested in reasoning only about deterministic or linear context-free programs,[3] or we might be interested only in a few special context-free programs such as $a^{\Delta}ba^{\Delta}$ or $a^{\Delta}b^{\Delta}$. Perhaps PDL remains decidable when these programs are added. The general question is to determine the borderline between the decidable and the undecidable when it comes to enriching the class of programs allowed in PDL.

Interestingly, if we wish to consider such simple nonregular extensions as $\mathsf{PDL}+a^{\Delta}ba^{\Delta}$ or $\mathsf{PDL}+a^{\Delta}b^{\Delta}$, we will not be able to prove undecidability by the technique used for context-free PDL in Theorem 26, since standard problems that are undecidable for context-free languages, such as equivalence and inclusion, are decidable for classes containing the regular languages and the likes of $a^{\Delta}ba^{\Delta}$ and $a^{\Delta}b^{\Delta}$. Moreover, we cannot prove decidability by the technique used for PDL in Section 3.2, since logics like $\mathsf{PDL}+a^{\Delta}ba^{\Delta}$ and $\mathsf{PDL}+a^{\Delta}b^{\Delta}$ do not enjoy the finite model property. Thus, if we want to determine the decidability status of such extensions, we will have to work harder.

THEOREM 27. *There is a satisfiable formula in* $\mathsf{PDL}+a^{\Delta}b^{\Delta}$ *that is not satisfied in any finite structure.*

[3] A *linear program* is one whose seqs are generated by a context-free grammar in which there is at most one nonterminal symbol on the right-hand side of each rule. This corresponds to a family of recursive procedures in which there is at most one recursive call in each procedure.

For $\mathsf{PDL} + a^{\Delta}ba^{\Delta}$, the news is worse than mere undecidability:

THEOREM 28. *The validity problem for* $\mathsf{PDL} + a^{\Delta}ba^{\Delta}$ *is* Π^1_1*-complete.*

The Π^1_1 result holds also for PDL extended with the two programs $a^{\Delta}b^{\Delta}$ and $b^{\Delta}a^{\Delta}$.

It is easy to show that the validity problem for context-free PDL in its entirety remains in Π^1_1. Together with the fact that $a^{\Delta}ba^{\Delta}$ is a context-free language, this yields an answer to the first question mentioned earlier: context-free PDL is Π^1_1-complete. As to the second question, Theorem 28 shows that the high undecidability phenomenon starts occurring even with the addition of one very simple nonregular program.

We now turn to nonregular programs over a single letter. Consider the language of powers of 2:

$$a^{2^*} \stackrel{\text{def}}{=} \{a^{2^i} \mid i \geq 0\}.$$

Here we have:

THEOREM 29. *The validity problem for* $\mathsf{PDL} + a^{2^*}$ *is undecidable.*

It is actually possible to prove this result for powers of any fixed $k \geq 2$. Thus PDL with the addition of any language of the form $\{a^{k^i} \mid i \geq 0\}$ for fixed $k \geq 2$ is undecidable. Another class of one-letter extensions that has been proven to be undecidable consists of Fibonacci-like sequences:

THEOREM 30. *Let* f_0, f_1 *be arbitrary elements of* $\mathbb{N}$ *with* $f_0 < f_1$*, and let* F *be the sequence* $f_0, f_1, f_2, \ldots$ *generated by the recurrence* $f_i = f_{i-1} + f_{i-2}$ *for* $i \geq 2$*. Let* $a^F \stackrel{\text{def}}{=} \{a^{f_i} \mid i \geq 0\}$*. Then the validity problem for* $\mathsf{PDL} + a^F$ *is undecidable.*

In both these theorems, the fact that the sequences of a's in the programs grow exponentially is crucial to the proofs. Indeed, we know of no undecidability results for any one-letter extension in which the lengths of the sequences of a's grow subexponentially. Particularly intriguing are the cases of squares and cubes:

$$a^{*^2} \stackrel{\text{def}}{=} \{a^{i^2} \mid i \geq 0\},$$
$$a^{*^3} \stackrel{\text{def}}{=} \{a^{i^3} \mid i \geq 0\}.$$

Are $\mathsf{PDL} + a^{*^2}$ and $\mathsf{PDL} + a^{*^3}$ undecidable?

There is a decidability result for a slightly restricted version of the squares extension, which seems to indicate that the full unrestricted version $\mathsf{PDL} + a^{*^2}$ is decidable too. However, we conjecture that for cubes the problem is undecidable. Interestingly, several classical open problems in number theory reduce to instances of the validity problem for $\mathsf{PDL} + a^{*^3}$. For example, while no one knows whether every integer greater than 10000 is the sum of five cubes, the following formula is valid if and only if the answer is yes:

$$[(a^{*^3})^5]p \quad \rightarrow \quad [a^{10001}a^*]p.$$

(The 5-fold and 10001-fold iterations have to be written out in full, of course.) If $\mathsf{PDL} + a^{*^3}$ were decidable, then we could compute the answer in a simple manner, at least in principle.

6.2 Decidable Extensions

We now turn to positive results. Theorem 27 states that $\mathsf{PDL} + a^{\Delta}b^{\Delta}$ does not have the finite model property. Nevertheless, we have the following:

THEOREM 31. *The validity problem for* $\mathsf{PDL} + a^{\Delta}b^{\Delta}$ *is decidable.*

When contrasted with Theorem 28, the decidability of $\mathsf{PDL} + a^{\Delta}b^{\Delta}$ is very surprising. We have two of the simplest nonregular languages—$a^{\Delta}ba^{\Delta}$ and $a^{\Delta}b^{\Delta}$—which are extremely similar, yet the addition of one to PDL yields high undecidability while the other leaves the logic decidable.

Theorem 31 was proved originally by showing that, although $\mathsf{PDL} + a^{\Delta}b^{\Delta}$ does not always admit finite models, it does admit finite *pushdown models*, in which transitions are labeled not only with atomic programs but also with push and pop instructions for a particular kind of stack. A close study of the proof (which relies heavily on the idiosyncrasies of the language $a^{\Delta}b^{\Delta}$) suggests that the decidability or undecidability has to do with the manner in which an automaton accepts the languages involved. For example, in the usual way of accepting $a^{\Delta}ba^{\Delta}$, a pushdown automaton (PDA) reading an a will carry out a push or a pop, depending upon its location in the input word. However, in the standard way of accepting $a^{\Delta}b^{\Delta}$, the a's are always pushed and the b's are always popped, regardless of the location; the input symbol alone determines what the automaton does. More recent work, which we now set out to describe, has yielded a general decidability result that confirms this intuition. It is of special interest due to its generality, since it does not depend on specific programs.

Let $M = (Q,\ \Sigma,\ \Gamma,\ q_0,\ z_0,\ \delta)$ be a PDA that accepts by empty stack. We say that M is *simple-minded* if, whenever $\delta(q, \sigma, \gamma) = (p, b)$, then for each q' and γ', either $\delta(q', \sigma, \gamma') = (p, b)$ or $\delta(q', \sigma, \gamma')$ is undefined. A context-free language is said to be *simple-minded* (a simple-minded CFL) if there exists a simple-minded PDA that accepts it.

In other words, the action of a simple-minded automaton is determined uniquely by the input symbol; the state and stack symbol are only used to help determine whether the machine halts (rejecting the input) or continues. Note that such an automaton is necessarily deterministic.

It is noteworthy that simple-minded PDAs accept a large fragment of the context-free languages, including $a^{\Delta}b^{\Delta}$ and $b^{\Delta}a^{\Delta}$, as well as all balanced parenthesis languages (Dyck sets) and many of their intersections with regular languages.

THEOREM 32. *If L is accepted by a simple-minded PDA, then* $\mathsf{PDL} + L$ *is decidable.*

We can obtain another general decidability result involving languages accepted by deterministic stack automata. A stack automaton is a one-way PDA whose head can travel up and down the stack reading its contents, but can make changes only at the top of the stack. Stack automata can accept non-context-free languages such as $a^\Delta b^\Delta c^\Delta$ and its generalizations $a_1^\Delta a_2^\Delta \ldots a_n^\Delta$ for any n, as well as many variants thereof. It would be nice to be able to prove decidability of PDL when augmented by any language accepted by such a machine, but this is not known. What has been proven, however, is that if each word in such a language is preceded by a new symbol to mark its beginning, then the enriched PDL is decidable:

THEOREM 33. *Let $e \notin \Pi_0$, and let L be a language over Π_0 that is accepted by a deterministic stack automaton. If we let eL denote the language $\{eu \mid u \in L\}$, then $\mathsf{PDL} + eL$ is decidable.*

While Theorems 32 and 33 are general and cover many languages, they do not prove decidability of $\mathsf{PDL} + a^\Delta b^\Delta c^\Delta$, which may be considered the simplest non-context-free extension of PDL. Nevertheless, the constructions used in the proofs of the two general results have been combined to yield:

THEOREM 34. *$\mathsf{PDL} + a^\Delta b^\Delta c^\Delta$ is decidable.*

As explained, we know of no undecidabile extension of PDL with a polynomially growing language, although we conjecture that the cubes extension is undecidable. Since the decidability status of such extensions seems hard to determine, we now address a weaker notion: the presence or absence of a finite model property. The technique used in Theorem 27 to show that $\mathsf{PDL} + a^\Delta b^\Delta$ violates the finite model property does not work for one-letter alphabets. Nevertheless, we now state a general result leading to many one-letter extensions that violate the finite model property. In particular, the theorem will yield the following:

PROPOSITION 35 (squares and cubes). *The logics $\mathsf{PDL} + a^{*^2}$ and $\mathsf{PDL} + a^{*^3}$ do not have the finite model property.*

PROPOSITION 36 (polynomials). *For every polynomial of the form*

$$p(n) \quad = \quad c_i n^i + c_{i-1} n^{i-1} + \cdots + c_0 \in \mathbb{Z}[n]$$

with $i \geq 2$ and positive leading coefficient $c_i > 0$, let $S_p = \{p(m) \mid m \in \mathbb{N}\} \cap \mathbb{N}$. Then $\mathsf{PDL} + a^{S_p}$ does not have the finite model property.

PROPOSITION 37 (sums of primes). *Let p_i be the i^{th} prime (with $p_1 = 2$), and define*

$$S_{\text{sop}} \quad \stackrel{\text{def}}{=} \quad \{\sum_{i=1}^{n} p_i \mid n \geq 1\}.$$

Then $\mathsf{PDL} + a^{S_{\text{sop}}}$ does not have the finite model property.

PROPOSITION 38 (factorials). *Let $S_{\text{fac}} \stackrel{\text{def}}{=} \{n! \mid n \in \mathbb{N}\}$. Then $\mathsf{PDL} + a^{S_{\text{fac}}}$ does not have the finite model property.*

The finite model property fails for any sufficiently fast-growing integer linear recurrence, not just the Fibonacci sequence, although we do not know whether these extensions also render PDL undecidable. A k^{th}*-order integer linear recurrence* is an inductively defined sequence

$$\ell_n \quad \stackrel{\mathrm{def}}{=} \quad c_1\ell_{n-1} + \cdots + c_k\ell_{n-k} + c_0, \quad n \geq k, \tag{12}$$

where $k \geq 1$, $c_0, \ldots, c_k \in \mathbb{N}$, $c_k \neq 0$, and $\ell_0, \ldots, \ell_{k-1} \in \mathbb{N}$ are given.

PROPOSITION 39 (linear recurrences). *Let* $S_{\mathrm{lr}} = \{\ell_n \mid n \geq 0\}$ *be the set defined inductively by* (12). *The following conditions are equivalent:*

(i) $a^{S_{\mathrm{lr}}}$ *is nonregular;*

(ii) $\mathsf{PDL} + a^{S_{\mathrm{lr}}}$ *does not have the finite model property;*

(iii) *not all* $\ell_0, \ldots, \ell_{k-1}$ *are zero and* $\sum_{i=1}^{k} c_i > 1$.

7 OTHER VARIANTS OF PDL

7.1 Deterministic Programs

Nondeterminism arises in PDL in two ways:

- atomic programs can be interpreted in a structure as (not necessarily single-valued) binary relations on states; and
- the programming constructs $\alpha \cup \beta$ and α^* involve nondeterministic choice.

Many modern programming languages have facilities for concurrency and distributed computation, certain aspects of which can be modeled by nondeterminism. Nevertheless, the majority of programs written in practice are still deterministic. Here we investigate the effect of eliminating either one or both of these sources of nondeterminism from PDL.

A program α is said to be (*semantically*) *deterministic* in a Kripke frame $\mathfrak{K}$ if its traces are uniquely determined by their first states. If α is an atomic program a, this is equivalent to the requirement that $\mathfrak{m}_{\mathfrak{K}}(a)$ be a partial function; that is, if both (s, t) and $(s, t') \in \mathfrak{m}_{\mathfrak{K}}(a)$, then $t = t'$. A *deterministic Kripke frame* $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ is one in which all atomic a are semantically deterministic.

The class of *deterministic* **while** *programs*, denoted DWP, is the class of programs in which

- the operators $\cup$, ?, and * may appear only in the context of the conditional test, **while** loop, **skip**, or **fail**;
- tests in the conditional test and **while** loop are purely propositional; that is, there is no occurrence of the < > or [ ] operators.

The class of *nondeterministic* **while** *programs*, denoted WP, is the same, except unconstrained use of the nondeterministic choice construct $\cup$ is allowed. It is easily shown that if α and β are semantically deterministic in $\mathfrak{K}$, then so are **if** φ **then** α **else** β and **while** φ **do** α.

By restricting either the syntax or the semantics or both, we obtain the following logics:

- DPDL (deterministic PDL), which is syntactically identical to PDL, but interpreted over deterministic structures only;
- SPDL (strict PDL), in which only deterministic **while** programs are allowed; and
- SDPDL (strict deterministic PDL), in which both restrictions are in force.

Validity and satisfiability in DPDL and SDPDL are defined just as in PDL, but with respect to deterministic structures only. If φ is valid in PDL, then φ is also valid in DPDL, but not conversely: the formula

$$<a>\varphi \quad \rightarrow \quad [a]\varphi \tag{13}$$

is valid in DPDL but not in PDL. Also, SPDL and SDPDL are strictly less expressive than PDL or DPDL, since the formula

$$<(a \cup b)^*>\varphi \tag{14}$$

is not expressible in SPDL, as shown in [Halpern and Reif, 1983].

THEOREM 40. *If the axiom scheme*

$$<a>\varphi \quad \rightarrow \quad [a]\varphi, \quad a \in \Pi_0 \tag{15}$$

is added to Axiom System 17, then the resulting system is sound and complete for DPDL.

THEOREM 41. *Validity in* DPDL *is deterministic exponential-time complete.*

Now we turn to SPDL, in which atomic programs can be nondeterministic but can be composed into larger programs only with deterministic constructs.

THEOREM 42. *Validity in* SPDL *is deterministic exponential-time complete.*

The final version of interest is SDPDL, in which both the syntactic restrictions of SPDL and the semantic ones of DPDL are adopted. The exponential-time lower bound fails here, and we have:

THEOREM 43. *The validity problem for* SDPDL *is complete in polynomial space.*

The question of relative power of expression is of interest here. Is DPDL $<$ PDL? Is SDPDL $<$ DPDL? The first of these questions is inappropriate, since the syntax of both languages is the same but they are interpreted over different classes of structures. Considering the second, we have:

THEOREM 44. SDPDL $<$ DPDL *and* SPDL $<$ PDL.

In summary, we have the following diagram describing the relations of expressiveness between these logics. The solid arrows indicate added expressive power and broken ones a difference in semantics. The validity problem is exponential-time complete for all but SDPDL, for which it is *PSPACE*-complete. Straightforward variants of Axiom System 17 are complete for all versions.

$$\begin{tikzcd}
& \mathsf{PDL} & \\
\mathsf{SPDL} \arrow[ur] & & \mathsf{DPDL} \arrow[ul, dashed] \\
& \mathsf{SDPDL} \arrow[ul, dashed] \arrow[ur] &
\end{tikzcd}$$

7.2 Representation by Automata

A PDL program represents a regular set of computation sequences. This same regular set could possibly be represented exponentially more succinctly by a finite automaton. The difference between these two representations corresponds roughly to the difference between **while** programs and flowcharts.

Since finite automata are exponentially more succinct in general, the upper bound of Section 5 could conceivably fail if finite automata were allowed as programs. Moreover, we must also rework the deductive system of Section 4.1.

However, it turns out that the completeness and exponential-time decidability results of PDL are not sensitive to the representation and still go through in the presence of finite automata as programs, provided the deductive system of Section 4.1 and the techniques of Sections 4 and 5 are suitably modified, as shown in [Pratt, 1979b; Pratt, 1981b] and [Harel and Sherman, 1985].

In recent years, the automata-theoretic approach to logics of programs has yielded significant insight into propositional logics more powerful than PDL, as well as substantial reductions in the complexity of their decision procedures. Especially enlightening are the connections with automata on infinite strings and infinite trees. By viewing a formula as an automaton and a treelike model as an input to that automaton, the satisfiability problem for a given formula becomes the emptiness problem for a given automaton. Logical questions are thereby transformed into purely automata-theoretic questions.

We assume that nondeterministic finite automata are given in the form

$$M \quad = \quad (n,\, i,\, j,\, \delta), \tag{16}$$

where $\overline{n} = \{0, \ldots, n-1\}$ is the set of states, $i, j \in \overline{n}$ are the start and final states respectively, and δ assigns a subset of $\Pi_0 \cup \{\varphi? \mid \varphi \in \Phi\}$ to each pair of states. Intuitively, when visiting state ℓ and seeing symbol a, the automaton may move to state k if $a \in \delta(\ell, k)$.

The fact that the automata (16) have only one accept state is without loss of generality. If M is an arbitrary nondeterministic finite automaton with accept states F, then the set accepted by M is the union of the sets accepted by M_k for $k \in F$, where M_k is identical to M except that it has unique accept state k. A desired formula $[M]\varphi$ can be written as a conjunction

$$\bigwedge_{k \in F} [M_k]\varphi$$

with at most quadratic growth.

We now obtain a new logic APDL (*automata* PDL) by defining Φ and Π inductively using the clauses for Φ from Section 2.1 and letting $\Pi = \Pi_0 \cup \{\varphi? \mid \varphi \in \Phi\} \cup F$, where F is the set of automata of the form (16).

Axioms 17(iv), (v), and (vii) are replaced by:

$$[n,i,j,\delta]\varphi \quad \leftrightarrow \quad \bigwedge_{\substack{k \in \overline{n} \\ \alpha \in \delta(i,k)}} [\alpha]\,[n,k,j,\delta]\varphi, \quad i \neq j \tag{17}$$

$$[n,i,i,\delta]\varphi \quad \leftrightarrow \quad \varphi \wedge \bigwedge_{\substack{k \in \overline{n} \\ \alpha \in \delta(i,k)}} [\alpha]\,[n,k,i,\delta]\varphi. \tag{18}$$

The induction axiom 17(viii) becomes

$$(\bigwedge_{k\in\overline{n}} [n,i,k,\delta](\varphi_k \ \to \bigwedge_{\substack{m\in\overline{n}\\ \alpha\in\delta(k,m)}} [\alpha]\varphi_m)) \ \to \ (\varphi_i \ \to \ [n,i,j,\delta]\varphi_j). \tag{19}$$

These and other similar changes can be used to prove:

THEOREM 45. *Validity in* APDL *is decidable in exponential time.*

THEOREM 46. *The axiom system described above is complete for* APDL.

7.3 Converse

The *converse operator* $^-$ is a program operator that allows a program to be "run backwards":

$$\mathfrak{m}_{\mathfrak{K}}(\alpha^-) \ \stackrel{\mathrm{def}}{=} \ \{(s,t) \mid (t,s) \in \mathfrak{m}_{\mathfrak{K}}(\alpha)\}.$$

PDL with converse is called CPDL.

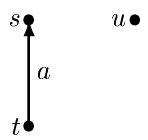
The following identities allow us to assume without loss of generality that the converse operator is applied to atomic programs only.

$$\begin{array}{rcl} (\alpha\,;\,\beta)^- & \leftrightarrow & \beta^-\,;\,\alpha^- \\ (\alpha\cup\beta)^- & \leftrightarrow & \alpha^-\cup\beta^- \\ \alpha^{*-} & \leftrightarrow & \alpha^{-*}. \end{array}$$

The converse operator strictly increases the expressive power of PDL, since the formula $<\alpha^->\mathbf{1}$ is not expressible without it.

THEOREM 47. PDL $<$ CPDL.

Proof. Consider the structure described in the following figure:



In this structure, $s \models <a^->\mathbf{1}$ but $u \not\models <a^->\mathbf{1}$. On the other hand, it can be shown by induction on the structure of formulas that if s and u agree on all atomic formulas, then no formula of PDL can distinguish between the two. ■

More interestingly, the presence of the converse operator implies that the operator $<\alpha>$ is *continuous* in the sense that if A is any (possibly infinite) family of formulas possessing a join $\bigvee A$, then $\bigvee <\alpha>A$ exists and is logically equivalent to $<\alpha>\bigvee A$. In the absence of the converse operator, one can construct nonstandard models for which this fails.

The completeness and exponential time decidability results of Sections 4 and 5 can be extended to CPDL provided the following two axioms are added:

$$\varphi \rightarrow [\alpha]\langle\alpha^-\rangle\varphi$$
$$\varphi \rightarrow [\alpha^-]\langle\alpha\rangle\varphi.$$

The filtration lemma (Lemma 14) still holds in the presence of $^-$, as does the finite model property.

7.4 Well-foundedness

If α is a deterministic program, the formula $\varphi \rightarrow \langle\alpha\rangle\psi$ asserts the total correctness of α with respect to pre- and postconditions φ and ψ, respectively. For *nondeterministic* programs, however, this formula does not express the right notion of total correctness. It asserts that φ implies that *there exists* a halting computation sequence of α yielding ψ, whereas we would really like to assert that φ implies that *all* computation sequences of α terminate and yield ψ. Let us denote the latter property by

$$TC(\varphi, \alpha, \psi).$$

Unfortunately, this is not expressible in PDL.

The problem is intimately connected with the notion of *well-foundedness*. A program α is said to be *well-founded* at a state u_0 if there exists no infinite sequence of states $u_0, u_1, u_2, \ldots$ with $(u_i, u_{i+1}) \in \mathfrak{m}_{\mathfrak{K}}(\alpha)$ for all $i \geq 0$. This property is not expressible in PDL either, as we will see.

Several very powerful logics have been proposed to deal with this situation. The most powerful is perhaps the propositional μ-calculus, which is essentially propositional modal logic augmented with a least fixpoint operator μ. Using this operator, one can express any property that can be formulated as the least fixpoint of a monotone transformation on sets of states defined by the PDL operators. For example, the well-foundedness of a program α is expressed

$$\mu X.[\alpha]X \tag{20}$$

in this logic.

Two somewhat weaker ways of capturing well-foundedness without resorting to the full μ-calculus have been studied. One is to add to PDL an explicit predicate **wf** for well-foundedness:

$$\mathfrak{m}_{\mathfrak{K}}(\mathbf{wf}\,\alpha) \stackrel{\text{def}}{=} \{s_0 \mid \neg\exists s_1, s_2, \ldots\ \forall i \geq 0\ (s_i, s_{i+1}) \in \mathfrak{m}_{\mathfrak{K}}(\alpha)\}.$$

Another is to add an explicit predicate **halt**, which asserts that all computations of its argument α terminate. The predicate **halt** can be defined inductively from **wf** as follows:

$$\mathbf{halt}\, a \stackrel{\text{def}}{\iff} \mathbf{1}, \quad a \text{ an atomic program or test,} \tag{21}$$

$$\mathbf{halt}\, \alpha;\beta \stackrel{\text{def}}{\iff} \mathbf{halt}\, \alpha \wedge [\alpha]\, \mathbf{halt}\, \beta, \tag{22}$$

$$\mathbf{halt}\, \alpha \cup \beta \stackrel{\text{def}}{\iff} \mathbf{halt}\, \alpha \wedge \mathbf{halt}\, \beta, \tag{23}$$

$$\mathbf{halt}\, \alpha^* \stackrel{\text{def}}{\iff} \mathbf{wf}\, \alpha \wedge [\alpha^*]\, \mathbf{halt}\, \alpha. \tag{24}$$

These constructs have been investigated under the various names **loop**, **repeat**, and Δ. The predicates **loop** and **repeat** are just the complements of **halt** and **wf**, respectively:

$$\mathbf{loop}\, \alpha \stackrel{\text{def}}{\iff} \neg\mathbf{halt}\, \alpha$$

$$\mathbf{repeat}\, \alpha \stackrel{\text{def}}{\iff} \neg\mathbf{wf}\, \alpha.$$

Clause (24) is equivalent to the assertion

$$\mathbf{loop}\, \alpha^* \quad \stackrel{\text{def}}{\iff} \quad \mathbf{repeat}\, \alpha \vee \langle\alpha^*\rangle \mathbf{loop}\, \alpha.$$

It asserts that a nonhalting computation of α^* consists of either an infinite sequence of halting computations of α or a finite sequence of halting computations of α followed by a nonhalting computation of α.

Let RPDL and LPDL denote the logics obtained by augmenting PDL with the **wf** and **halt** predicates, respectively.[4] It follows from the preceding discussion that

$$\mathsf{PDL} \quad \leq \quad \mathsf{LPDL} \quad \leq \quad \mathsf{RPDL} \quad \leq \quad \text{the propositional } \mu\text{-calculus.}$$

Moreover, all these inclusions are known to be strict.

The logic LPDL is powerful enough to express the total correctness of nondeterministic programs. The total correctness of α with respect to precondition φ and postcondition ψ is expressed

$$TC(\varphi, \alpha, \psi) \quad \stackrel{\text{def}}{\iff} \quad \varphi \;\rightarrow\; \mathbf{halt}\, \alpha \wedge [\alpha]\, \psi.$$

Conversely, **halt** can be expressed in terms of TC:

$$\mathbf{halt}\, \alpha \quad \Leftrightarrow \quad TC(\mathbf{1}, \alpha, \mathbf{1}).$$

THEOREM 48. PDL $<$ LPDL.

THEOREM 49. LPDL $<$ RPDL.

It is possible to extend Theorem 49 to versions CRPDL and CLPDL in which converse is allowed in addition to **wf** or **halt**. Also, the proof of Theorem 47 goes

[4] The L in LPDL stands for "loop" and the R in RPDL stands for "repeat." We retain these names for historical reasons.

through for LPDL and RPDL, so that $\texttt{<}a^-\texttt{>}\mathbf{1}$ is not expressible in either. Theorem 48 goes through for the converse versions too. We obtain the situation illustrated in the following figure, in which the arrows indicate $<$ and the absence of a path between two logics means that each can express properties that the other cannot.

$$
\begin{tikzcd}
 & & \mathsf{CRPDL} & \\
 & \mathsf{CLPDL} \arrow[ur] & & \mathsf{RPDL} \arrow[ul] \\
\mathsf{CPDL} \arrow[ur] & & \mathsf{LPDL} \arrow[ul] \arrow[ur] & \\
 & \mathsf{PDL} \arrow[ul] \arrow[ur] & &
\end{tikzcd}
$$

The filtration lemma fails for all **halt** and **wf** versions as in Theorem 48. However, satisfiable formulas of the μ-calculus (hence of RPDL and LPDL) do have finite models. This finite model property is not shared by CLPDL or CRPDL.

THEOREM 50. *The* CLPDL *formula*

$$\neg\textbf{\textit{halt}}\, a^* \wedge \texttt{[}a^*\texttt{]}\,\textbf{\textit{halt}}\, a^{*-}$$

is satisfiable but has no finite model.

As it turns out, Theorem 50 does not prevent CRPDL from being decidable.

THEOREM 51. *The validity problems for* CRPDL*,* CLPDL*,* RPDL*,* LPDL*, and the propositional* μ*-calculus are all decidable in deterministic exponential time.*

Obviously, the simpler the logic, the simpler the arguments needed to show exponential time decidability. Over the years all these logics have been gradually shown to be decidable in exponential time by various authors using various techniques. Here we point to the exponential time decidability of the propositional μ-calculus with forward and backward modalities, proved in [Vardi, 1998b], from which all these can be seen easily to follow. The proof in [Vardi, 1998b] is carried out by exhibiting an exponential time decision procedure for two-way alternating automata on infinite trees.

As mentioned above, RPDL possesses the finite (but not necessarily the small and not the collapsed) model property.

THEOREM 52. *Every satisfiable formula of* RPDL*,* LPDL*, and the propositional* μ*-calculus has a finite model.*

CRPDL and CLPDL are extensions of PDL that, like PDL $+\, a^{\Delta}b^{\Delta}$ (Theorems 27 and 31), are decidable despite lacking a finite model property.

Complete axiomatizations for RPDL and LPDL can be obtained by embedding them into the μ-calculus (see Section 14.4).

7.5 Concurrency

Another interesting extension of PDL concerns concurrent programs. One can define an intersection operator $\cap$ such that the binary relation on states corresponding to the program $\alpha \cap \beta$ is the intersection of the binary relations corresponding to α and β. This can be viewed as a kind of concurrency operator that admits transitions to those states that both α and β would have admitted.

Here we consider a different and perhaps more natural notion of concurrency. The interpretation of a program will not be a binary relation on states, which relates initial states to possible final states, but rather a relation between a states and *sets* of states. Thus $\mathfrak{m}_{\mathfrak{K}}(\alpha)$ will relate a start state u to a collection of sets of states U. The intuition is that starting in state u, the (concurrent) program α can be run with its concurrent execution threads ending in the set of final states U. The basic concurrency operator will be denoted here by $\wedge$, although in the original work on concurrent Dynamic Logic ([Peleg, 1987b; Peleg, 1987c; Peleg, 1987a]) the notation $\cap$ is used.

The syntax of *concurrent* PDL is the same as PDL, with the addition of the clause:

- if $\alpha, \beta \in \Pi$, then $\alpha \wedge \beta \in \Pi$.

The program $\alpha \wedge \beta$ means intuitively, "Execute α and β in parallel."

The semantics of concurrent PDL is defined on Kripke frames $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ as with PDL, except that for programs α,

$$\mathfrak{m}_{\mathfrak{K}}(\alpha) \quad \subseteq \quad K \times 2^{K}.$$

Thus the meaning of α is a collection of *reachability pairs* of the form (u, U), where $u \in K$ and $U \subseteq K$. In this brief description of concurrent PDL, we require that structures assign to atomic programs *sequential*, non-parallel, meaning; that is, for each $a \in \Pi_0$, we require that if $(u, U) \in \mathfrak{m}_{\mathfrak{K}}(a)$, then $\#U = 1$. The true parallelism will stem from applying the concurrency operator to build larger sets U in the reachability pairs of compound programs. For details, see [Peleg, 1987b; Peleg, 1987c].

The relevant results for this logic are the following:

THEOREM 53. PDL $<$ *concurrent* PDL.

THEOREM 54. *The validity problem for concurrent* PDL *is decidable in deterministic exponential time.*

Axiom System 17, augmented with the following axiom, can be be shown to be complete for concurrent PDL:

$$<\alpha \land \beta>\varphi \quad \leftrightarrow \quad <\alpha>\varphi \land <\beta>\varphi.$$

8 FIRST-ORDER DYNAMIC LOGIC (DL)

In this section we begin the study of first-order Dynamic Logic. The main difference between first-order DL and the propositional version PDL discussed in previous sections is the presence of a first-order structure $\mathfrak{A}$, called the *domain of computation*, over which first-order quantification is allowed. States are no longer abstract points, but *valuations* of a set of variables over A, the *carrier* of $\mathfrak{A}$. Atomic programs in DL are no longer abstract binary relations, but *assignment statements* of various forms, all based on assigning values to variables during the computation. The most basic example of such an assignment is the *simple assignment* $x := t$, where x is a variable and t is a term. The atomic formulas of DL are generally taken to be atomic first-order formulas.

In addition to the constructs of PDL, the basic DL syntax contains individual variables ranging over A, function and predicate symbols for distinguished functions and predicates of $\mathfrak{A}$, and quantifiers ranging over A, exactly as in classical first-order logic. More powerful versions of the logic contain array and stack variables and other constructs, as well as primitive operations for manipulating them, and assignments for changing their values. Sometimes the introduction of a new construct increases expressive power and sometimes not; sometimes it has an effect on the complexity of deciding satisfiability and sometimes not. Indeed, one of the central goals of research has been to classify these constructs in terms of their relative expressive power and complexity.

In this section we lay the groundwork for this by defining the various logical and programming constructs we shall need.

8.1 Basic Syntax

The language of first-order Dynamic Logic is built upon classical first-order logic. There is always an underlying first-order vocabulary Σ, which involves a vocabulary of function symbols and predicate (or relation) symbols. On top of this vocabulary, we define a set of *programs* and a set of *formulas*. These two sets interact by means of the modal construct [ ] exactly as in the propositional case. Programs and formulas are usually defined by mutual induction.

Let $\Sigma = \{f, g, \ldots, p, r, \ldots\}$ be a finite first-order vocabulary. Here f and g denote typical function symbols of Σ, and p and r denote typical relation symbols. Associated with each function and relation symbol of Σ is a fixed *arity* (number of arguments), although we do not represent the arity explicitly. We assume that Σ always contains the equality symbol $=$, whose arity is 2. Functions and relations of arity $0, 1, 2, 3$ and n are called *nullary*, *unary*, *binary*, *ternary*, and *n-ary*, respectively. Nullary functions are also called *constants*. We shall be using a countable set of *individual variables* $V = \{x_0, x_1, \ldots\}$.

We always assume that Σ contains at least one function symbol of positive arity. A vocabulary Σ is *polyadic* if it contains a function symbol of arity greater than one. Vocabularies whose function symbols are all unary are called *monadic*.

A vocabulary Σ is *rich* if either it contains at least one predicate symbol besides the equality symbol or the sum of arities of the function symbols is at least two. Examples of rich vocabularies are: two unary function symbols, or one binary function symbol, or one unary function symbol and one unary predicate symbol. A vocabulary that is not rich is *poor*. Hence a poor vocabulary has just one unary function symbol and possibly some constants, but no relation symbols other than equality. The main difference between rich and poor vocabularies is that the former admit exponentially many pairwise non-isomorphic structures of a given finite cardinality, whereas the latter admit only polynomially many.

We say that the vocabulary Σ is *mono-unary* if it contains no function symbols other than a single unary one. It may contain constants and predicate symbols.

The definitions of DL programs and formulas below depend on the vocabulary Σ, but in general we shall not make this dependence explicit unless we have some specific reason for doing so.

Atomic Formulas and Programs

In all versions of DL that we will consider, atomic formulas are atomic formulas of the first-order vocabulary Σ; that is, formulas of the form $r(t_1, \ldots, t_n)$, where r is an n-ary relation symbol of Σ and $t_1, \ldots, t_n$ are terms of Σ.

As in PDL, programs are defined inductively from atomic programs using various programming constructs. The meaning of a compound program is given inductively in terms of the meanings of its constituent parts. Different classes of programs are obtained by choosing different classes of atomic programs and programming constructs.

In the basic version of DL, an atomic program is a *simple assignment* $x := t$, where $x \in V$ and t is a term of Σ. Intuitively, this program assigns the value of t to the variable x. This is the same form of assignment found in most conventional programming languages.

More powerful forms of assignment such as stack and array assignments and nondeterministic "wildcard" assignments will be discussed later. The precise choice of atomic programs will be made explicit when needed, but for now, we use the term *atomic program* to cover all of these possibilities.

Tests

As in PDL, DL contains a test operator $?$, which turns a formula into a program. In most versions of DL that we shall discuss, we allow only quantifier-free first-order formulas as tests. We sometimes call these versions *poor test*. Alternatively, we might allow any first-order formula as a test. Most generally, we might place no restrictions on the form of tests, allowing any DL formula whatsoever, including those that contain other programs, perhaps containing other tests, *etc*. These versions of DL are labeled *rich test* as in Section 2.1. Whereas programs can be defined independently from formulas in poor test versions, rich test versions require a mutually inductive definition of programs and formulas.

As with atomic programs, the precise logic we consider at any given time depends on the choice of tests we allow. We will make this explicit when needed, but for now, we use the term *test* to cover all possibilities.

Regular Programs

For a given set of atomic programs and tests, the set of *regular programs* is defined as in PDL (see Section 2.1):

- any atomic program or test is a program;
- if α and β are programs, then $\alpha\,;\,\beta$ is a program;
- if α and β are programs, then $\alpha \cup \beta$ is a program;
- if α is a program then α^* is a program.

While *Programs*

Much of the literature on DL is concerned with the class of **while** *programs* (see Section 2.1). Formally, *deterministic* **while** *programs* form the subclass of the regular programs in which the program operators $\cup$, ?, and * are constrained to appear only in the forms

$$\begin{aligned} \textbf{skip} &\stackrel{\text{def}}{=} \mathbf{1}? \\ \textbf{fail} &\stackrel{\text{def}}{=} \mathbf{0}? \\ \textbf{if } \varphi \textbf{ then } \alpha \textbf{ else } \beta &\stackrel{\text{def}}{=} (\varphi?;\alpha) \cup (\neg\varphi?;\beta) \qquad (25) \\ \textbf{while } \varphi \textbf{ do } \alpha &\stackrel{\text{def}}{=} (\varphi?;\alpha)^*;\neg\varphi? \qquad (26) \end{aligned}$$

The class of *nondeterministic* **while** *programs* is the same, except that we allow unrestricted use of the nondeterministic choice construct $\cup$. Of course, unrestricted use of the sequential composition operator is allowed in both languages.

Restrictions on the form of atomic programs and tests apply as with regular programs. For example, if we are allowing only poor tests, then the φ occurring in the programs (25) and (26) must be a quantifier-free first-order formula.

The class of deterministic **while** programs is important because it captures the basic programming constructs common to many real-life imperative programming languages. Over the standard structure of the natural numbers $\mathbb{N}$, deterministic **while** programs are powerful enough to define all partial recursive functions, and thus over $\mathbb{N}$ they are as as expressive as regular programs. A similar result holds for a wide class of models similar to $\mathbb{N}$, for a suitable definition of "partial recursive functions" in these models. However, it is not true in general that **while** programs, even nondeterministic ones, are universally expressive. We discuss these results in Section 12.

Formulas

A *formula* of DL is defined in way similar to that of PDL, with the addition of a rule for quantification. Equivalently, we might say that a formula of DL is defined in a way similar to that of first-order logic, with the addition of a rule for modality. The basic version of DL is defined with regular programs:

- the false formula **0** is a formula;
- any atomic formula is a formula;
- if φ and ψ are formulas, then $\varphi \to \psi$ is a formula;
- if φ is a formula and $x \in V$, then $\forall x\ \varphi$ is a formula;
- if φ is a formula and α is a program, then $[\alpha]\varphi$ is a formula.

The only missing rule in the definition of the syntax of DL are the tests. In our basic version we would have:

- if φ is a quatifier-free first-order formula, then $\varphi?$ is a test.

For the rich test version, the definitions of programs and formulas are mutually dependent, and the rule defining tests is simply:

- if φ is a formula, then $\varphi?$ is a test.

We will use the same notation as in propositional logic that $\neg\varphi$ stands for $\varphi \to \mathbf{0}$. As in first-order logic, the first-order existential quantifier $\exists$ is considered a defined construct: $\exists x\ \varphi$ abbreviates $\neg\forall x\ \neg\varphi$. Similarly, the modal construct < > is considered a defined construct as in Section 2.1, since it is the modal dual of [ ]. The other propositional constructs $\wedge$, $\vee$, $\leftrightarrow$ are defined as in Section 2.1. Of course, we use parentheses where necessary to ensure unique readability.

Note that the individual variables in V serve a dual purpose: they are both program variables and logical variables.

8.2 Richer Programs

Seqs and R.E. Programs

Some classes of programs are most conveniently defined as certain sets of seqs. Recall from Section 2.3 that a *seq* is a program of the form $\sigma_1; \cdots ; \sigma_k$, where each σ_i is an assignment statement or a quantifier-free first-order test. Each regular program α is associated with a unique set of seqs $CS(\alpha)$ (Section 2.3). These definitions were made in the propositional context, but they apply equally well to the first-order case; the only difference is in the form of atomic programs and tests.

Construing the word in the broadest possible sense, we can consider a *program* to be an arbitrary set of seqs. Although this makes sense semantically—we can assign an input/output relation to such a set in a meaningful way—such programs can hardly be called executable. At the very least we should require that the set of seqs be recursively enumerable, so that there will be some effective procedure that can list all possible executions of a given program. However, there is a subtle issue that arises with this notion. Consider the set of seqs

$$\{x_i := f^i(c) \mid i \in \mathbb{N}\}.$$

This set satisfies the above restriction, yet it can hardly be called a program. It uses infinitely many variables, and as a consequence it might change a valuation at infinitely many places. Another pathological example is the set of seqs

$$\{x_{i+1} := f(x_i) \mid i \in \mathbb{N}\},$$

which not only could change a valuation at infinitely many locations, but also depends on infinitely many locations of the input valuation.

In order to avoid such pathologies, we will require that each program use only finitely many variables. This gives rise to the following definition of *r.e. programs*, which is the most general family of programs we will consider. Specifically, an r.e. program α is a Turing machine that enumerates a set of seqs over a finite set of variables. The set of seqs enumerated will be called $CS(\alpha)$. By $FV(\alpha)$ we will denote the finite set of variables that occur in seqs of $CS(\alpha)$.

An important issue connected with r.e. programs is that of *bounded memory*. The assignment statements or tests in an r.e. program may have infinitely many terms with increasingly deep nesting of function symbols (although, as discussed, these terms only use finitely many variables), and these could require an unbounded amount of memory to compute. We define a set of seqs to be *bounded memory* if the depth of terms appearing in it is bounded. In fact, without sacrificing computational power, we could require that all terms be of the form $f(x_1, \ldots, x_n)$ in a bounded-memory set of seqs.

Arrays and Stacks

Interesting variants of the programming language we use in DL arise from allowing auxiliary data structures. We shall define versions with *arrays* and *stacks*, as well as a version with a nondeterministic assignment statement called *wildcard assignment*.

Besides these, one can imagine augmenting **while** programs with many other kinds of constructs such as blocks with declarations, recursive procedures with various parameter passing mechanisms, higher-order procedures, concurrent processes, *etc*. It is easy to arrive at a family consisting of thousands of programming languages, giving rise to thousands of logics. Obviously, we have had to restrict ourselves. It is worth mentioning, however, that certain kinds of recursive procedures are captured by our stack operations, as explained below.

Arrays

To handle arrays, we include a countable set of *array variables*

$$V_{\text{array}} \quad = \quad \{F_0, F_1, \dots\}.$$

Each array variable has an associated *arity*, or number of arguments, which we do not represent explicitly. We assume that there are countably many variables of each arity $n \geq 0$. In the presence of array variables, we equate the set V of individual variables with the set of nullary array variables; thus $V \subseteq V_{\text{array}}$.

The variables in V_{array} of arity n will range over n-ary functions with arguments and values in the domain of computation. In our exposition, elements of the domain of computation play two roles: they are used both as *indices* into an array and as *values* that can be stored in an array. One might equally well introduce a separate sort for array indices; although conceptually simple, this would complicate the notation and would give no new insight.

We extend the set of first-order terms to allow the unrestricted occurrence of array variables, provided arities are respected.

The classes of *regular programs with arrays* and *deterministic* and *nondeterministic* **while** *programs with arrays* are defined similarly to the classes without, except that we allow *array assignments* in addition to simple assignments. Array assignments are similar to simple assignments, but on the left-hand side we allow a term in which the outermost symbol is an array variable:

$$F(t_1, \dots, t_n) := t.$$

Here F is an n-ary array variable and $t_1, \dots, t_n, t$ are terms, possibly involving other array variables. Note that when $n = 0$, this reduces to the ordinary simple assignment.

Recursion via an Algebraic Stack

We now consider DL in which the programs can manipulate a stack. The literature in automata theory and formal languages often distinguishes a stack from a pushdown store. In the former, the automaton is allowed to inspect the contents of the stack but to make changes only at the top. We shall use the term stack to denote the more common pushdown store, where the only inspection allowed is at the top of the stack.

The motivation for this extension is to be able to capture recursion. It is well known that recursive procedures can be modeled using a stack, and for various technical reasons we prefer to extend the data-manipulation capabilities of our programs than to introduce new control constructs. When it encounters a recursive call, the stack simulation of recursion will push the return location and values of local variables and parameters on the stack. It will pop them upon completion of the call. The LIFO (last-in-first-out) nature of stack storage fits the order in which control executes recursive calls.

To handle the stack in our stack version of DL, we add two new atomic programs

$$\textbf{push}(t) \quad \text{and} \quad \textbf{pop}(y),$$

where t is a term and $y \in V$. Intuitively, **push**(t) pushes the current value of t onto the top of the stack, and **pop**(y) pops the top value off the top of the stack and assigns that value to the variable y. If the stack is empty, the pop operation does not change anything. We could have added a test for stack emptiness, but it can be shown to be redundant. Formally, the stack is simply a finite string of elements of the domain of computation.

The classes of *regular programs with stack* and *deterministic* and *nondeterministic* **while** *programs with stack* are obtained by augmenting the respective classes of programs with the **push** and **pop** operations as atomic programs in addition to simple assignments.

In contrast to the case of arrays, here there is only a single stack. In fact, expressiveness changes dramatically when two or more stacks are allowed. Also, in order to be able to simulate recursion, the domain must have at least two distinct elements so that return addresses can be properly encoded in the stack. One way of doing this is to store the return address itself in unary using one element of the domain, then store one occurrence of the second element as a delimiter symbol, followed by domain elements constituting the current values of parameters and local variables.

The kind of stack described here is often termed *algebraic*, since it contains elements from the domain of computation. It should be contrasted with the Boolean stack described next.

Parameterless Recursion via a Boolean Stack

An interesting special case is when the stack can contain only two distinct elements. This version of our programming language can be shown to capture recursive procedures without parameters or local variables. This is because we only need to store return addresses, but no actual data items from the domain of computation. This can be achieved using two values, as described above. We thus arrive at the idea of a Boolean stack.

To handle such a stack in this version of DL, we add three new kinds of atomic programs and one new test. The atomic programs are

$$\textbf{push-1} \qquad \textbf{push-0} \qquad \textbf{pop},$$

and the test is simply **top**?. Intuitively, **push-1** and **push-0** push the corresponding distinct Boolean values on the stack, **pop** removes the top element, and the test **top**? evaluates to true iff the top element of the stack is $\mathbf{1}$, but with no side effect.

With the test `top`? only, there is no explicit operator that distinguishes a stack with top element **0** from the empty stack. We might have defined such an operator, and in a more realistic language we would certainly do so. However, it is mathematically redundant, since it can be simulated with the operators we already have.

Wildcard Assignment

The nondeterministic assignment $x := ?$ is a device that arises in the study of fairness; see [Apt and Plotkin, 1986]. It has often been called *random assignment* in the literature, although it has nothing to do with randomness or probability. We shall call it *wildcard assignment*. Intuitively, it operates by assigning a nondeterministically chosen element of the domain of computation to the variable x. This construct together with the $[\,]$ modality is similar to the first-order universal quantifier, since it will follow from the semantics that the two formulas $[x := ?]\varphi$ and $\forall x\ \varphi$ are equivalent. However, wildcard assignment may appear in programs and can therefore be iterated.

8.3 Semantics

In this section we assign meanings to the syntactic constructs described in the previous sections. We interpret programs and formulas over a first-order structure $\mathfrak{A}$. Variables range over the carrier of this structure. We take an *operational* view of program semantics: programs change the values of variables by sequences of simple assignments $x := t$ or other assignments, and flow of control is determined by the truth values of tests performed at various times during the computation.

States as Valuations

An instantaneous snapshot of all relevant information at any moment during the computation is determined by the values of the program variables. Thus our *states* will be *valuations* $u, v, \ldots$ of the variables V over the carrier of the structure $\mathfrak{A}$. Our formal definition will associate the pair (u, v) of such valuations with the program α if it is possible to start in valuation u, execute the program α, and halt in valuation v. In this case, we will call (u, v) an *input/output pair* of α and write $(u, v) \in \mathfrak{m}_{\mathfrak{A}}(\alpha)$. This will result in a Kripke frame exactly as in Section 2.

Let $\mathfrak{A} = (A,\ \mathfrak{m}_{\mathfrak{A}})$ be a first-order structure for the vocabulary Σ. We call $\mathfrak{A}$ the *domain of computation*. Here A is a set, called the *carrier* of $\mathfrak{A}$, and $\mathfrak{m}_{\mathfrak{A}}$ is a *meaning function* such that $\mathfrak{m}_{\mathfrak{A}}(f)$ is an n-ary function $\mathfrak{m}_{\mathfrak{A}}(f) : A^n \to A$ interpreting the n-ary function symbol f of Σ, and $\mathfrak{m}_{\mathfrak{A}}(r)$ is an n-ary relation $\mathfrak{m}_{\mathfrak{A}}(r) \subseteq A^n$ interpreting the n-ary relation symbol r of Σ. The equality symbol $=$ is always interpreted as the identity relation.

For $n \geq 0$, let $A^n \to A$ denote the set of all n-ary functions in A. By convention, we take $A^0 \to A = A$. Let A^* denote the set of all finite-length strings over A.

The structure $\mathfrak{A}$ determines a Kripke frame, which we will also denote by $\mathfrak{A}$, as follows. A *valuation* over $\mathfrak{A}$ is a function u assigning an n-ary function over A to each n-ary array variable. It also assigns meanings to the stacks as follows. We shall use the two unique variable names STK and $BSTK$ to denote the algebraic stack and the Boolean stack, respectively. The valuation u assigns a finite-length string of elements of A to STK and a finite-length string of Boolean values **1** and **0** to $BSTK$. Formally:

$$\begin{array}{rcll} u(F) & \in & A^n \to A, & \text{if } F \text{ is an } n\text{-ary array variable,} \\ u(STK) & \in & A^*, & \\ u(BSTK) & \in & \{\mathbf{1},\mathbf{0}\}^*. & \end{array}$$

By our convention $A^0 \to A = A$, and assuming that $V \subseteq V_{\text{array}}$, the individual variables (that is, the nullary array variables) are assigned elements of A under this definition:

$$u(x) \in A \text{ if } x \in V.$$

The valuation u extends uniquely to terms t by induction. For an n-ary function symbol f and an n-ary array variable F,

$$\begin{array}{rcl} u(f(t_1,\ldots,t_n)) & \stackrel{\text{def}}{=} & \mathfrak{m}_{\mathfrak{A}}(f)(u(t_1),\ldots,u(t_n)) \\ u(F(t_1,\ldots,t_n)) & \stackrel{\text{def}}{=} & u(F)(u(t_1),\ldots,u(t_n)). \end{array}$$

The *function-patching* operator is defined as follows: if X and D are sets, $f : X \to D$ is any function, $x \in X$, and $d \in D$, then $f[x/d] : X \to D$ is the function defined by

$$f[x/d](y) \quad \stackrel{\text{def}}{=} \quad \begin{cases} d, & \text{if } x = y \\ f(y), & \text{otherwise.} \end{cases}$$

We will be using this notation in several ways, both at the logical and metalogical levels. For example:

- If u is a valuation, x is an individual variable, and $a \in A$, then $u[x/a]$ is the new valuation obtained from u by changing the value of x to a and leaving the values of all other variables intact.
- If F is an n-ary array variable and $f : A^n \to A$, then $u[F/f]$ is the new valuation that assigns the same value as u to the stack variables and to all array variables other than F, and

$$u[F/f](F) \quad = \quad f.$$

- If $f : A^n \to A$ is an n-ary function and $\overline{a} = a_1,\ldots,a_n \in A^n$ and $a \in A$, then the expression $f[\overline{a}/a]$ denotes the n-ary function that agrees with f everywhere except for input $\overline{a}$, on which it takes the value a. More precisely,

$$f[\overline{a}/a](\overline{b}) \quad = \quad \begin{cases} a, & \text{if } \overline{b} = \overline{a} \\ f(\overline{b}), & \text{otherwise.} \end{cases}$$

We call valuations u and v *finite variants* of each other if

$$u(F)(a_1, \dots, a_n) \quad = \quad v(F)(a_1, \dots, a_n)$$

for all but finitely many array variables F and n-tuples $a_1, \dots, a_n \in A^n$. In other words, u and v differ on at most finitely many array variables, and for those F on which they do differ, the functions $u(F)$ and $v(F)$ differ on at most finitely many values.

The relation "is a finite variant of" is an equivalence relation on valuations. Since a halting computation can run for only a finite amount of time, it can execute only finitely many assignments. It will therefore not be able to cross equivalence class boundaries; that is, in the binary relation semantics given below, if the pair (u, v) is an input/output pair of the program α, then v is a finite variant of u.

We are now ready to define the *states* of our Kripke frame. For $a \in A$, let w_a be the valuation in which the stacks are empty and all array and individual variables are interpreted as constant functions taking the value a everywhere. A *state* of $\mathfrak{A}$ is any finite variant of a valuation w_a. The set of states of $\mathfrak{A}$ is denoted $S^{\mathfrak{A}}$.

Call a state *initial* if it differs from some w_a only at the values of individual variables.

It is meaningful, and indeed useful in some contexts, to take as states the set of *all* valuations. Our purpose in restricting our attention to states as defined above is to prevent arrays from being initialized with highly complex oracles that would compromise the value of the relative expressiveness results of Section 12.

Assignment Statements

As in Section 2.2, with every program α we associate a binary relation

$$\mathfrak{m}_{\mathfrak{A}}(\alpha) \quad \subseteq \quad S^{\mathfrak{A}} \times S^{\mathfrak{A}}$$

(called the *input/output relation* of p), and with every formula φ we associate a set

$$\mathfrak{m}_{\mathfrak{A}}(\varphi) \quad \subseteq \quad S^{\mathfrak{A}}.$$

The sets $\mathfrak{m}_{\mathfrak{A}}(\alpha)$ and $\mathfrak{m}_{\mathfrak{A}}(\varphi)$ are defined by mutual induction on the structure of α and φ.

For the basis of this inductive definition, we first give the semantics of all the assignment statements discussed earlier.

- The array assignment $F(t_1, \dots, t_n) := t$ is interpreted as the binary relation

  $$\begin{aligned} &\mathfrak{m}_{\mathfrak{A}}(F(t_1, \dots, t_n) := t) \\ &\quad \stackrel{\text{def}}{=} \quad \{(u, u[F/u(F)[u(t_1), \dots, u(t_n)/u(t)]]) \mid u \in S^{\mathfrak{A}}\}. \end{aligned}$$

  In other words, starting in state u, the array assignment has the effect of changing the value of F on input $u(t_1), \dots, u(t_n)$ to $u(t)$, and leaving the value of F on all other inputs and the values of all other variables intact. For $n = 0$, this definition reduces to the following definition of simple assignment:

$$\mathfrak{m}_{\mathfrak{A}}(x := t) \quad \stackrel{\text{def}}{=} \quad \{(u, u[x/u(t)]) \mid u \in S^{\mathfrak{A}}\}.$$

- The push operations, **push**(t) for the algebraic stack and **push-1** and **push-0** for the Boolean stack, are interpreted as the binary relations

$$\begin{aligned} \mathfrak{m}_{\mathfrak{A}}(\textbf{push}(t)) &\stackrel{\text{def}}{=} \{(u, u[STK/(u(t) \cdot u(STK))]) \mid u \in S^{\mathfrak{A}}\} \\ \mathfrak{m}_{\mathfrak{A}}(\textbf{push-1}) &\stackrel{\text{def}}{=} \{(u, u[BSTK/(\mathbf{1} \cdot u(BSTK))]) \mid u \in S^{\mathfrak{A}}\} \\ \mathfrak{m}_{\mathfrak{A}}(\textbf{push-0}) &\stackrel{\text{def}}{=} \{(u, u[BSTK/(\mathbf{0} \cdot u(BSTK))]) \mid u \in S^{\mathfrak{A}}\}, \end{aligned}$$

respectively. In other words, **push**(t) changes the value of the algebraic stack variable STK from $u(STK)$ to the string $u(t) \cdot u(STK)$, the concatenation of the value $u(t)$ with the string $u(STK)$, and everything else is left intact. The effects of **push-1** and **push-0** are similar, except that the special constants **1** and **0** are concatenated with $u(BSTK)$ instead of $u(t)$.

- The pop operations, **pop**(y) for the algebraic stack and **pop** for the Boolean stack, are interpreted as the binary relations

$$\begin{aligned} &\mathfrak{m}_{\mathfrak{A}}(\textbf{pop}(y)) \\ &\quad \stackrel{\text{def}}{=} \{(u, u[STK/\textbf{tail}(u(STK))][y/\textbf{head}(u(STK), u(y))]) \mid u \in S^{\mathfrak{A}}\} \\ &\mathfrak{m}_{\mathfrak{A}}(\textbf{pop}) \\ &\quad \stackrel{\text{def}}{=} \{(u, u[BSTK/\textbf{tail}(u(BSTK))]) \mid u \in S^{\mathfrak{A}}\}, \end{aligned}$$

respectively, where

$$\begin{aligned} \textbf{tail}(a \cdot \sigma) &\stackrel{\text{def}}{=} \sigma \\ \textbf{tail}(\varepsilon) &\stackrel{\text{def}}{=} \varepsilon \\ \textbf{head}(a \cdot \sigma, b) &\stackrel{\text{def}}{=} a \\ \textbf{head}(\varepsilon, b) &\stackrel{\text{def}}{=} b \end{aligned}$$

and ε is the empty string. In other words, if $u(STK) \neq \varepsilon$, this operation changes the value of STK from $u(STK)$ to the string obtained by deleting the first element of $u(STK)$ and assigns that element to the variable y. If $u(STK) = \varepsilon$, then nothing is changed. Everything else is left intact. The Boolean stack operation **pop** changes the value of $BSTK$ only, with no additional changes. We do not include explicit constructs to test whether the stacks are empty, since these can be simulated. However, we do need to be able to refer to the value of the top element of the Boolean stack, hence we include the $\mathtt{top}$? test.

- The Boolean test program $\mathbf{top}?$ is interpreted as the binary relation

$$\mathfrak{m}_{\mathfrak{A}}(\mathbf{top}?) \quad \stackrel{\text{def}}{=} \quad \{(u,u) \mid u \in S^{\mathfrak{A}},\ \mathbf{head}(u(BSTK)) = \mathbf{1}\}.$$

  In other words, this test changes nothing at all, but allows control to proceed iff the top of the Boolean stack contains $\mathbf{1}$.

- The wildcard assignment $x := ?$ for $x \in V$ is interpreted as the relation

$$\mathfrak{m}_{\mathfrak{A}}(x := ?) \quad \stackrel{\text{def}}{=} \quad \{(u, u[x/a]) \mid u \in S^{\mathfrak{A}},\ a \in A\}.$$

  As a result of executing this statement, x will be assigned some arbitrary value of the carrier set A, and the values of all other variables will remain unchanged.

Programs and Formulas

The meanings of compound programs and formulas are defined by mutual induction on the structure of α and φ essentially as in the propositional case (see Section 2.2). We include these definitions below for completeness.

Regular Programs and **While** *Programs*

Here are the semantic definitions for the four constructs of regular programs.

$$\begin{aligned}
\mathfrak{m}_{\mathfrak{A}}(\alpha\,;\,\beta) &\stackrel{\text{def}}{=} \mathfrak{m}_{\mathfrak{A}}(\alpha) \circ \mathfrak{m}_{\mathfrak{A}}(\beta) \\
&= \{(u,v) \mid \exists w\ (u,w) \in \mathfrak{m}_{\mathfrak{A}}(\alpha) \text{ and } (w,v) \in \mathfrak{m}_{\mathfrak{A}}(\beta)\} && (27)\\
\mathfrak{m}_{\mathfrak{A}}(\alpha \cup \beta) &\stackrel{\text{def}}{=} \mathfrak{m}_{\mathfrak{A}}(\alpha) \cup \mathfrak{m}_{\mathfrak{A}}(\beta) && (28)\\
\mathfrak{m}_{\mathfrak{A}}(\alpha^*) &\stackrel{\text{def}}{=} \mathfrak{m}_{\mathfrak{A}}(\alpha)^* \ = \ \bigcup_{n \geq 0} \mathfrak{m}_{\mathfrak{A}}(\alpha)^n \\
\mathfrak{m}_{\mathfrak{A}}(\varphi?) &\stackrel{\text{def}}{=} \{(u,u) \mid u \in \mathfrak{m}_{\mathfrak{A}}(\varphi)\}. && (29)
\end{aligned}$$

The semantics of defined constructs such as **if-then-else** and **while-do** are obtained using their definitions exactly as in PDL.

Seqs and R.E. Programs

Recall that an r.e. program is a Turing machine enumerating a set $CS(\alpha)$ of seqs. If α is an r.e. program, we define

$$\mathfrak{m}_{\mathfrak{A}}(\alpha) \quad \stackrel{\text{def}}{=} \quad \bigcup_{\sigma \in CS(\alpha)} \mathfrak{m}_{\mathfrak{A}}(\sigma).$$

Thus, the meaning of α is defined to be the union of the meanings of the seqs in $CS(\alpha)$. The meaning $\mathfrak{m}_{\mathfrak{A}}(\sigma)$ of a seq σ is determined by the meanings of atomic programs and tests and the sequential composition operator.

There is an interesting point here regarding the translation of programs using other programming constructs into r.e. programs. This can be done for arrays and stacks (for Booleans stacks, even into r.e. programs with bounded memory), but not for wildcard assignment. Since later in the book we shall be referring to the r.e. set of seqs associated with such programs, it is important to be able to carry out this translation. To see how this is done for the case of arrays, for example, consider an algorithm for simulating the execution of a program by generating only ordinary assignments and tests. It does not generate an array assignment of the form $F(t_1, \ldots, t_n) := t$, but rather "remembers" it and when it reaches an assignment of the form $x := F(t_1, \ldots, t_n)$ it will aim at generating $x := t$ instead. This requires care, since we must keep track of changes in the variables inside t and $t_1, \ldots, t_n$ and incorporate them into the generated assignments.

Formulas

Here are the semantic definitions for the constructs of formulas of DL. The semantics of atomic first-order formulas is the standard semantics of classical first-order logic.

$$\mathfrak{m}_{\mathfrak{A}}(\mathbf{0}) \stackrel{\text{def}}{=} \varnothing \tag{30}$$

$$\mathfrak{m}_{\mathfrak{A}}(\varphi \to \psi) \stackrel{\text{def}}{=} \{u \mid \text{if } u \in \mathfrak{m}_{\mathfrak{A}}(\varphi) \text{ then } u \in \mathfrak{m}_{\mathfrak{A}}(\psi)\} \tag{31}$$

$$\mathfrak{m}_{\mathfrak{A}}(\forall x\, \varphi) \stackrel{\text{def}}{=} \{u \mid \forall a \in A\; u[x/a] \in \mathfrak{m}_{\mathfrak{A}}(\varphi)\} \tag{32}$$

$$\mathfrak{m}_{\mathfrak{A}}([\alpha]\varphi) \stackrel{\text{def}}{=} \{u \mid \forall v \text{ if } (u,v) \in \mathfrak{m}_{\mathfrak{A}}(\alpha) \text{ then } v \in \mathfrak{m}_{\mathfrak{A}}(\varphi)\}. \tag{33}$$

Equivalently, we could define the first-order quantifiers $\forall$ and $\exists$ in terms of the wildcard assignment:

$$\forall x\, \varphi \leftrightarrow [x := ?]\varphi \tag{34}$$

$$\exists x\, \varphi \leftrightarrow \langle x := ?\rangle\varphi. \tag{35}$$

Note that for *deterministic* programs α (for example, those obtained by using the **while** programming language instead of regular programs and disallowing wildcard assignments), $\mathfrak{m}_{\mathfrak{A}}(\alpha)$ is a partial function from states to states; that is, for every state u, there is at most one v such that $(u,v) \in \mathfrak{m}_{\mathfrak{A}}(\alpha)$. The partiality of the function arises from the possibility that α may not halt when started in certain states. For example, $\mathfrak{m}_{\mathfrak{A}}(\textbf{while 1 do skip})$ is the empty relation. In general, the relation $\mathfrak{m}_{\mathfrak{A}}(\alpha)$ need not be single-valued.

If K is a given set of syntactic constructs, we refer to the version of Dynamic Logic with programs built from these constructs as *Dynamic Logic with* K or simply as $\mathsf{DL}(K)$. Thus, we have $\mathsf{DL}(\text{r.e.})$, $\mathsf{DL}(\text{array})$, $\mathsf{DL}(\text{stk})$, $\mathsf{DL}(\text{bstk})$,

$\mathsf{DL}(\mathrm{wild})$, and so on. As a default, these logics are the poor-test versions, in which only quantifier-free first-order formulas may appear as tests. The unadorned DL is used to abbreviate $\mathsf{DL}(\mathrm{reg})$, and we use $\mathsf{DL}(\mathrm{dreg})$ to denote DL with **while** programs, which are really deterministic regular programs. Again, **while** programs use only poor tests. Combinations such as DL(dreg+wild) are also allowed.

8.4 Satisfiability and Validity

The concepts of satisfiability, validity, *etc*. are defined as for PDL in Section 2 or as for first-order logic under the standard semantics.

Let $\mathfrak{A} = (A, \mathfrak{m}_{\mathfrak{A}})$ be a structure, and let u be a state in $S^{\mathfrak{A}}$. For a formula φ, we write $\mathfrak{A}, u \vDash \varphi$ if $u \in \mathfrak{m}_{\mathfrak{A}}(\varphi)$ and say that u *satisfies* φ *in* $\mathfrak{A}$. We sometimes write $u \vDash \varphi$ when $\mathfrak{A}$ is understood. We say that φ is $\mathfrak{A}$-*valid* and write $\mathfrak{A} \vDash \varphi$ if $\mathfrak{A}, u \vDash \varphi$ for all u in $\mathfrak{A}$. We say that φ is *valid* and write $\vDash \varphi$ if $\mathfrak{A} \vDash \varphi$ for all $\mathfrak{A}$. We say that φ is *satisfiable* if $\mathfrak{A}, u \vDash \varphi$ for some $\mathfrak{A}, u$.

For a set of formulas Δ, we write $\mathfrak{A} \vDash \Delta$ if $\mathfrak{A} \vDash \varphi$ for all $\varphi \in \Delta$.

Informally, $\mathfrak{A}, u \vDash [\alpha]\varphi$ iff every terminating computation of α starting in state u terminates in a state satisfying φ, and $\mathfrak{A}, u \vDash <\alpha>\varphi$ iff there exists a computation of α starting in state u and terminating in a state satisfying φ. For a pure first-order formula φ, the metastatement $\mathfrak{A}, u \vDash \varphi$ has the same meaning as in first-order logic.

9 RELATIONSHIPS WITH STATIC LOGICS

9.1 Uninterpreted Reasoning

In contrast to the propositional version PDL discussed in Sections 2–7, DL formulas involve variables, functions, predicates, and quantifiers, a state is a mapping from variables to values in some domain, and atomic programs are assignment statements. To give semantic meaning to these constructs requires a first-order structure $\mathfrak{A}$ over which to interpret the function and predicate symbols. Nevertheless, we are not obliged to assume anything special about $\mathfrak{A}$ or the nature of the interpretations of the function and predicate symbols, except as dictated by first-order semantics. Any conclusions we draw from this level of reasoning will be valid under all possible interpretations. *Uninterpreted reasoning* refers to this style of reasoning.

For example, the formula

$$p(f(x), g(y, f(x))) \quad \rightarrow \quad \texttt{<}z := f(x)\texttt{>}p(z, g(y, z))$$

is true over any domain, irrespective of the interpretations of p, f, and g.

Another example of a valid formula is

$$\begin{aligned}&z = y \ \wedge \ \forall x\ f(g(x)) = x\\ &\quad \rightarrow \quad \texttt{[}\mathbf{while}\ p(y)\ \mathbf{do}\ y := g(y)\texttt{]}\texttt{<}\mathbf{while}\ y \neq z\ \mathbf{do}\ y := f(y)\texttt{>}\mathbf{1}.\end{aligned}$$

Note the use of [ ] applied to < >. This formula asserts that under the assumption that f "undoes" g, any computation consisting of applying g some number of times to z can be backtracked to the original z by applying f some number of times to the result.

We now observe that three basic properties of classical (uninterpreted) first-order logic, the *Löwenheim–Skolem theorem*, *completeness*, and *compactness*, fail for even fairly weak versions of DL.

The Löwenheim–Skolem theorem for classical first-order logic states that if a formula φ has an infinite model then it has models of all infinite cardinalities. Because of this theorem, classical first-order logic cannot define the structure of elementary arithmetic

$$\mathbb{N} \quad = \quad (\omega,\, +,\, \cdot,\, 0,\, 1,\, =)$$

up to isomorphism. That is, there is no first-order sentence that is true in a structure $\mathfrak{A}$ if and only if $\mathfrak{A}$ is isomorphic to $\mathbb{N}$. However, this can be done in DL.

PROPOSITION 55. *There exists a formula* $\Theta_{\mathbb{N}}$ *of* $\mathsf{DL}(\mathrm{dreg})$ *that defines* $\mathbb{N}$ *up to isomorphism.*

The Löwenheim–Skolem theorem does not hold for DL, because $\Theta_{\mathbb{N}}$ has an infinite model (namely $\mathbb{N}$), but all models are isomorphic to $\mathbb{N}$ and are therefore countable.

Besides the Löwenheim–Skolem Theorem, compactness fails in DL as well. Consider the following countable set Γ of formulas:

$$\{<\textbf{while}\ p(x)\ \textbf{do}\ x := f(x)>\mathbf{1}\} \quad \cup \quad \{p(f^n(x)) \mid n \geq 0\}.$$

It is easy to see that Γ is not satisfiable, but it is finitely satisfiable, *i.e.* each finite subset of it is satisfiable.

Worst of all, completeness cannot hold for any deductive system as we normally think of it (a finite effective system of axioms schemes and finitary inference rules). The set of theorems of such a system would be r.e., since they could be enumerated by writing down the axioms and systematically applying the rules of inference in all possible ways. However, the set of valid statements of DL is not recursively enumerable. In fact, we will describe in Section 10 exactly how bad the situation is.

This is not to say that we cannot say anything meaningful about proofs and deduction in DL. On the contrary, there is a wealth of interesting and practical results on axiom systems for DL that we will cover in Section 11.

In this section we investigate the power of DL relative to classical static logics on the uninterpreted level. In particular, *rich test* DL *of r.e. programs* is equivalent to the infinitary language $L_{\omega_1^{\mathrm{ck}}\omega}$. Some consequences of this fact are drawn in later sections.

First we introduce a definition that allows to compare different variants of DL. Let us recall from Section 8.3 that a state is *initial* if it differs from a constant state w_a only at the values of individual variables. If DL_1 and DL_2 are two variants of DL over the same vocabulary, we say that DL_2 is *as expressive as* DL_1 and write $\mathsf{DL}_1 \leq \mathsf{DL}_2$ if for each formula φ in DL_1 there is a formula ψ in DL_2 such that $\mathfrak{A}, u \vDash \varphi \leftrightarrow \psi$ for all structures $\mathfrak{A}$ and initial states u. If DL_2 is as expressive as DL_1 but DL_1 is not as expressive as DL_2, we say that DL_2 is *strictly more expressive than* DL_1, and write $\mathsf{DL}_1 < \mathsf{DL}_2$. If DL_2 is as expressive as DL_1 and DL_1 is as expressive as DL_2, we say that DL_1 and DL_2 are of *equal expressive power*, or are simply *equivalent*, and write $\mathsf{DL}_1 \equiv \mathsf{DL}_2$. We will also use these notions for comparing versions of DL with static logics such as $L_{\omega\omega}$.

There is a technical reason for the restriction to initial states in the above definition. If DL_1 and DL_2 have access to different sets of data types, then they may be trivially incomparable for uninteresting reasons, unless we are careful to limit the states on which they are compared. We shall see examples of this in Section 12.

Also, in the definition of $\mathsf{DL}(K)$ given in Section 8.4, the programming language K is an explicit parameter. Actually, the particular first-order vocabulary Σ over which $\mathsf{DL}(K)$ and K are considered should be treated as a parameter too. It turns out that the relative expressiveness of versions of DL is sensitive not only to K, but also to Σ. This second parameter is often ignored in the literature, creating a source of potential misinterpretation of the results. For now, we assume a fixed first-order vocabulary Σ.

Rich Test Dynamic Logic of R.E. Programs

We are about to introduce the most general version of DL we will ever consider. This logic is called *rich test Dynamic Logic of r.e. programs*, and it will be denoted $\mathsf{DL}(\text{rich-test r.e.})$. Programs of $\mathsf{DL}(\text{rich-test r.e.})$ are r.e. sets of seqs as defined in Section 8.2, except that the seqs may contain tests $\varphi?$ for any previously constructed formula φ.

The formal definition is inductive. All atomic programs are programs and all atomic formulas are formulas. If φ, ψ are formulas, α, β are programs, $\{\alpha_n \mid n \in \omega\}$ is an r.e. set of programs over a finite set of variables (free or bound), and x is a variable, then

- **0**
- $\varphi \to \psi$
- $[\alpha]\varphi$
- $\forall x\, \varphi$

are formulas and

- $\alpha\, ;\, \beta$
- $\{\alpha_n \mid n \in \omega\}$
- $\varphi?$

are programs. The set $CS(\alpha)$ of computation sequences of a rich test r.e. program α is defined as usual.

The language $L_{\omega_1\omega}$ is the language with the formation rules of the first-order language $L_{\omega\omega}$, but in which countably infinite conjunctions and disjunctions $\bigwedge_{i\in I}\varphi_i$ and $\bigvee_{i\in I}\varphi_i$ are also allowed. In addition, if $\{\varphi_i \mid i \in I\}$ is recursively enumerable, then the resulting language is denoted $L_{\omega_1^{\mathrm{ck}}\omega}$ and is sometimes called *constructive* $L_{\omega_1\omega}$.

PROPOSITION 56. $\mathsf{DL}(\text{rich-test r.e.}) \equiv L_{\omega_1^{\mathrm{ck}}\omega}$.

Since r.e. programs as defined in Section 8.2 are clearly a special case of general rich-test r.e. programs, it follows that $\mathsf{DL}(\text{rich-test r.e.})$ is as expressive as $\mathsf{DL}(\text{r.e.})$. In fact they are not of the same expressive power.

THEOREM 57. $\mathsf{DL}(\text{r.e.}) < \mathsf{DL}(\text{rich-test r.e.})$.

Henceforth, we shall assume that the first-order vocabulary Σ contains at least one function symbol of positive arity. Under this assumption, DL can easily be shown to be strictly more expressive than $L_{\omega\omega}$:

THEOREM 58. $L_{\omega\omega} < \mathsf{DL}$.

COROLLARY 59.

$$L_{\omega\omega} \quad < \quad \mathsf{DL} \quad < \quad \mathsf{DL}(\text{r.e.}) \quad < \quad \mathsf{DL}(\text{rich-test r.e.}) \quad \equiv \quad L_{\omega_1^{\mathrm{ck}}\omega}.$$

The situation with the intermediate versions of DL, e.g. $\mathsf{DL}(\mathrm{stk})$, $\mathsf{DL}(\mathrm{bstk})$, $\mathsf{DL}(\mathrm{wild})$, *etc*., is of interest. We deal with the relative expressive power of these in Section 12.

9.2 *Interpreted Reasoning*

Arithmetical Structures

This is the most detailed level we will consider. It is the closest to the actual process of reasoning about concrete, fully specified programs. Syntactically, the programs and formulas are as on the uninterpreted level, but here we assume a fixed structure or class of structures.

In this framework, we can study programs whose computational behavior depends on (sometimes deep) properties of the particular structures over which they are interpreted. In fact, almost any task of verifying the correctness of an actual program falls under the heading of interpreted reasoning.

One specific structure we will look at carefully is the natural numbers with the usual arithemetic operations:

$$\mathbb{N} \quad = \quad (\omega,\ 0,\ 1,\ +,\ \cdot,\ =).$$

Let $-$ denote the (first-order-definable) operation of subtraction and let $\mathrm{gcd}(x, y)$ denote the first-order-definable operation giving the greatest common divisor of x and y. The following formula of DL is $\mathbb{N}$-valid, i.e., true in all states of $\mathbb{N}$:

$$x = x' \land y = y' \land xy \geq 1 \quad \rightarrow \quad < \alpha > (x = \mathrm{gcd}(x', y')) \tag{36}$$

where α is the **while** program of Example 1 or the regular program

$$(x \neq y?; ((x > y?; x := x - y) \cup (x < y?; y := y - x)))^{*} x = y?.$$

Formula (36) states the correctness and termination of an actual program over $\mathbb{N}$ computing the greatest common divisor.

As another example, consider the following formula over $\mathbb{N}$:

$$\forall x \geq 1 < (\textbf{if } \mathbf{even}(x) \textbf{ then } x := x/2 \textbf{ else } x := 3x + 1)^{*} > (x = 1).$$

Here $/$ denotes integer division, and $\mathbf{even}(\ )$ is the relation that tests if its argument is even. Both of these are first-order definable. This innocent-looking formula asserts that starting with an arbitrary positive integer and repeating the following two operations, we will eventually reach 1:

- if the number is even, divide it by 2;
- if the number is odd, triple it and add 1.

The truth of this formula is as yet unknown, and it constitutes a problem in number theory (dubbed "the $3x + 1$ problem") that has been open for over 60 years. The formula $\forall x \geq 1$ $<\alpha>\mathbf{1}$, where α is

$$\textbf{while } x \neq 1 \textbf{ do if } \mathbf{even}(x) \textbf{ then } x := x/2 \textbf{ else } x := 3x + 1,$$

says this in a slightly different way.

The specific structure $\mathbb{N}$ can be generalized, resulting in the class of *arithmetical structures*. Briefly, a structure $\mathfrak{A}$ is *arithmetical* if it contains a first-order-definable copy of $\mathbb{N}$ and has first-order definable functions for coding finite sequences of elements of $\mathfrak{A}$ into single elements and for the corresponding decoding.

Arithmetical structures are important because (i) most structures arising naturally in computer science (*e.g.*, discrete structures with recursively defined data types) are arithmetical, and (ii) any structure can be extended to an arithmetical one by adding appropriate encoding and decoding capabilities. While most of the results we present for the interpreted level are given in terms of $\mathbb{N}$ alone, many of them hold for any arithmetical structure, so their significance is greater.

Expressive Power over $\mathbb{N}$

The results of Corollary 59 establishing that

$$L_{\omega\omega} \quad < \quad \mathsf{DL} \quad < \quad \mathsf{DL}(\text{r.e.}) \quad < \quad \mathsf{DL}(\text{rich-test r.e.})$$

were on the uninterpreted level, where all structures are taken into account. Thus first-order logic, regular DL, and $\mathsf{DL}(\text{rich-test r.e.})$ form a sequence of increasingly more powerful logics when interpreted uniformly over all structures.

What happens if one fixes a structure, say $\mathbb{N}$? Do these differences in expressive power still hold? We now address these questions.

First, we introduce notation for comparing expressive power over $\mathbb{N}$. If DL_1 and DL_2 are variants of DL (or static logics, such as $L_{\omega\omega}$) and are defined over the vocabulary of $\mathbb{N}$, we write $\mathsf{DL}_1 \leq_{\mathbb{N}} \mathsf{DL}_2$ if for each $\varphi \in \mathsf{DL}_1$ there is $\psi \in \mathsf{DL}_2$ such that $\mathbb{N} \vDash \varphi \leftrightarrow \psi$. We define $<_{\mathbb{N}}$ and $\equiv_{\mathbb{N}}$ from $\leq_{\mathbb{N}}$ in a way analogous to the definition of $<$ and $\equiv$ from $\leq$.

It turns out that over $\mathbb{N}$, DL is no more expressive than first-order logic $L_{\omega\omega}$. This is true even for finite-test DL. The result is stated for $\mathbb{N}$, but is actually true for any arithmetical structure.

THEOREM 60. $L_{\omega\omega} \equiv_{\mathbb{N}} \mathsf{DL} \equiv_{\mathbb{N}} \mathsf{DL}(\text{r.e.})$.

The significance of this result is that in principle, one can carry out all reasoning about programs interpreted over $\mathbb{N}$ in the first-order logic $L_{\omega\omega}$ by translating each DL formula into an equivalent first-order formula. The translation is effective. Moreover, Theorem 60 holds for any arithmetical structure containing the requisite coding power. As mentioned earlier, every structure can be extended to an arithmetical one.

However, the translation of Theorem 60 produces unwieldly formulas having little resemblance to the original ones. This mechanism is thus somewhat unnatural and does not correspond closely to the type of arguments one would find in practical program verication. In Section 11, a remedy is provided that makes the process more orderly.

We now observe that over $\mathbb{N}$, $\mathsf{DL}(\text{rich-test r.e.})$ has considerably more power than the equivalent logics of Theorem 60. This too is true for any arithmetical structure.

THEOREM 61. *Over* $\mathbb{N}$, $\mathsf{DL}(\text{rich-test r.e.})$ *defines precisely the* Δ_1^1 *(hyperarithmetic) sets.*

Theorems 60 and 61 say that over $\mathbb{N}$, the languages DL and $\mathsf{DL}(\text{r.e.})$ define the arithmetic (first-order definable) sets and $\mathsf{DL}(\text{rich-test r.e.})$ defines the hyperarithmetic or Δ_1^1 sets. Since the inclusion between these classes is strict—for example, first-order number theory is hyperarithmetic but not arithmetic—we have

COROLLARY 62. $\mathsf{DL}(\text{r.e.}) <_{\mathbb{N}} \mathsf{DL}(\text{rich-test r.e.})$.

10 COMPLEXITY OF DL

This section addresses the complexity of first-order Dynamic Logic.

Since all versions of DL subsume first-order logic, the truth, satisfiability, or validity of a given formula can be no easier to establish than in $L_{\omega\omega}$. Also, since DL(r.e.) is subsumed by $L_{\omega_1^{\mathrm{ck}}\omega}$, these questions are no harder to establish than in $L_{\omega_1^{\mathrm{ck}}\omega}$. These bounds hold for both uninterpreted and interpreted levels of reasoning.

10.1 The Uninterpreted Level

In this section we discuss the complexity of the validity problem for DL. By the remarks above, this problem is between Σ_1^0 and Π_1^1. That is, as a lower bound it is undecidable and can be no better than recursively enumerable, and as an upper bound it is in Π_1^1. This is a rather large gap, so we are still interested in determining more precise complexity bounds for DL and its variants. An interesting related question is whether there is some nontrivial[5] fragment of DL that is in Σ_1^0, since this would allow a complete axiomatization.

In the following, we consider these questions for full DL(reg), but we also consider two important subclasses of formulas for which better upper bounds are derivable:

- partial correctness assertions of the form $\psi \to [\alpha]\,\varphi$, and
- termination or total correctness assertions of the form $\psi \to \langle\alpha\rangle\varphi$,

where φ and ψ are first-order formulas. The results are stated for regular programs, but they remain true for the more powerful programming languages too. They also hold for deterministic **while** programs.

We state the results without mentioning the underlying first-order vocabulary Σ. For the upper bounds this is irrelevant. For the lower bounds, we assume the Σ contains a unary function symbol and ternary predicate symbols.

THEOREM 63. *The validity problem for* DL *is Π_1^1-hard, even for formulas of the form $\exists x\ [\alpha]\,\varphi$, where α is a regular program and φ is first-order.*

THEOREM 64. *The validity problem for* DL *and* DL(rich-test r.e.)*, as well as all intermediate versions, is Π_1^1-complete.*

To soften the negative flavor of these results, we now observe that the special cases of unquantified one-program DL(r.e.) formulas have easier validity problems (though, as mentioned, they are still undecidable).

THEOREM 65. *The validity problem for the sublanguage of* DL(r.e.) *consisting of formulas of the form $\langle\alpha\rangle\varphi$, where φ is first-order and α is an r.e. program, is Σ_1^0-complete.*

---

[5] *Nontrivial* here means containing $L_{\omega\omega}$ and allowing programs with iteration. The reason for this requirement is that loop-free programs add no expressive power over first-order logic.

It is easy to see that the result holds for formulas of the form $\psi \to \langle\alpha\rangle\varphi$, where ψ is also first-order. Thus, termination assertions for nondeterministic programs with first-order tests (or total correctness assertions for deterministic programs), on the uninterpreted level of reasoning, are recursively enumerable and therefore axiomatizable. We shall give an explicit axiomatization in Section 11.

We now turn to partial correctness.

THEOREM 66. *The validity problem for the sublanguage of* $\mathsf{DL}(\mathrm{r.e.})$ *consisting of formulas of the form* $[\alpha]\varphi$*, where* φ *is first-order and* α *is an r.e. program, is* Π^0_2*-complete. The* Π^0_2*-completeness property holds even if we restrict* α *to range over deterministic* **while** *programs.*

Theorem 66 extends easily to partial correctness assertions; that is, to formulas of the form $\psi \to [\alpha]\varphi$, where ψ is also first-order. Thus, while Π^0_2 is obviously better than Π^1_1, it is noteworthy that on the uninterpreted level of reasoning, the truth of even simple correctness assertions for simple programs is not r.e., so that no finitary complete axiomatization for such validities can be given.

10.2 The Interpreted Level

The characterizations of the various versions of DL in terms of classical static logics established in Section 9.2 provide us with the precise complexity of the validity problem over $\mathbb{N}$.

THEOREM 67. *The* $\mathbb{N}$*-validity problem for* $\mathsf{DL}(\mathrm{dreg})$ *and* $\mathsf{DL}(\text{rich-test r.e.})$*, as well as all intermediate versions, when defined over the vocabulary of* $\mathbb{N}$*, is hyperarithmetic (*Δ^1_1*) but not arithmetic.*

10.3 Spectral Complexity

We now introduce the *spectral complexity* of a programming language. As mentioned, this notion provides a measure of the complexity of the halting problem for programs over finite interpretations.

Recall that a *state* is a finite variant of a constant valuation w_a for some $a \in A$ (see Section 8.3), and a state w is *initial* if it differs from w_a for individual variables only. Thus, an initial state can be uniquely defined by specifying its relevant portion of values on individual variables. For $m \in \mathbb{N}$, we call an initial state w an m*-state* if for some $a \in A$ and for all $i \geq m$, $w(x_i) = a$. An m-state can be specified by an $(m+1)$-tuple of values $(a_0, \ldots, a_m)$ that represent values of w for the first $m+1$ individual variables $x_0, \ldots, x_m$. Call an m-state $w = (a_0, \ldots, a_m)$ *Herbrand-like* if the set $\{a_0, \ldots, a_m\}$ generates A; that is, if every element of A can be obtained as a value of a term in the state w.

We are now ready to define the notion of a *spectrum* of a programming language. Let K be a programming language and let $\alpha \in K$ and $m \geq 0$. The m^{th}

spectrum of α is the set

$$SP_m(\alpha) \stackrel{\mathrm{def}}{=} \{\ulcorner \mathfrak{A}_w \urcorner \mid \mathfrak{A} \text{ is a finite } \Sigma\text{-structure, } w \text{ is an } m\text{-state in } \mathfrak{A}\text{, and } \mathfrak{A}, w \models \langle\alpha\rangle\mathbf{1}\}.$$

The *spectrum* of K is the set

$$SP(K) \stackrel{\mathrm{def}}{=} \{SP_m(\alpha) \mid \alpha \in K,\ m \in \mathbb{N}\}.$$

Given $m \geq 0$, observe that structures in $S_n^{\Sigma \cup \{c_0, \ldots, c_m\}}$ can be viewed as structures of the form $\mathfrak{A}_w$ for a certain Σ-structure $\mathfrak{A}$ and an m-state w in $\mathfrak{A}$. This representation is unique.

In this section we establish the complexity of spectra; that is, the complexity of the halting problem in finite interpretations. Let us fix $m \geq 0$, a rich vocabulary Σ, and new constants $c_0, \ldots, c_m$. Since not every binary string is of the form $\ulcorner \mathfrak{A} \urcorner$ for some Σ-structure $\mathfrak{A}$ and m-state w in $\mathfrak{A}$, we will restrict our attention to strings that are of this form. Let

$$H_m^\Sigma \stackrel{\mathrm{def}}{=} \{\ulcorner \mathfrak{A} \urcorner \mid \mathfrak{A} \in S_n^{\Sigma \cup \{c_0, \ldots, c_m\}} \text{ for some } n \geq 1\}.$$

It is easy to show that the language H_m^Σ is in $LOGSPACE$ for every vocabulary Σ and $m \geq 0$.

We are now ready to connect complexity classes with spectra. Let K be any programming language and let $C \subseteq 2^{\{0,1\}^*}$ be a family of sets. We say that $SP(K)$ *captures* C, denoted $SP(K) \approx C$, if

- $SP(K) \subseteq C$, and
- for every $X \in C$ and $m \geq 0$, if $X \subseteq H_m^\Sigma$, then there is a program $\alpha \in K$ such that $SP_m(\alpha) = X$.

For example, if C is the class of all sets recognizable in polynomial time, then $SP(K) \approx P$ means that

- the halting problem over finite interpretations for programs from K is decidable in polynomial time, and
- every polynomial-time-recognizable set of codes of finite interpretations is the spectrum of some program from K.

We conclude this section by characterizing the spectral complexity of some of the programming languages introduced in Section 8.

THEOREM 68. *Let* Σ *be a rich vocabulary. Then*

(i) $SP(\mathrm{dreg}) \subseteq LOGSPACE$.

(ii) $SP(\mathrm{reg}) \subseteq NLOGSPACE$.

Moreover, if Σ *is mono-unary, then* $SP(\mathrm{dreg})$ *captures* $LOGSPACE$ *and* $SP(\mathrm{reg})$ *captures* $NLOGSPACE$.

THEOREM 69. *Over a rich vocabulary* Σ, $SP(\mathrm{dstk})$ *and* $SP(\mathrm{stk})$ *capture P*.

THEOREM 70. *If* Σ *is a rich vocabulary, then* $SP(\mathrm{darray})$ *and* $SP(\mathrm{array})$ *capture PSPACE.*

11 AXIOMATIZATION OF DL

11.1 Uninterpreted Reasoning

Recall from Section 10.1 that validity in DL is Π^1_1-complete, but only r.e. when restricted to simple termination assertions. This means that termination (or total correctness when the programs are deterministic) can be fully axiomatized in the standard sense. This we do first, and we then turn to the problem of axiomatizing full DL.

Since the validity problem for such termination assertions is r.e., it is of interest to find a nicely-structured complete axiom system. We propose the following.

Axiom System S1

Axiom Schemes

- all instances of valid first-order formulas;
- all instances of valid formulas of PDL;
- $\varphi[x/t] \to \mathord{<}x := t\mathord{>}\varphi$, where φ is a first-order formula.

Inference Rules

- modus ponens:

$$\frac{\varphi,\ \varphi \to \psi}{\psi}$$

We denote provability in Axiom System S1 by $\vdash_{S1}$.

THEOREM 71. *For any* DL *formula of the form* $\varphi \to \mathord{<}\alpha\mathord{>}\psi$*, for first-order* φ *and* ψ *and program* α *containing first-order tests only,*

$$\models \varphi \to \mathord{<}\alpha\mathord{>}\psi \quad \Leftrightarrow \quad \vdash_{S1} \varphi \to \mathord{<}\alpha\mathord{>}\psi.$$

Given the high undecidability of validity in DL, we cannot hope for a complete axiom system in the usual sense. Nevertheless, we do want to provide an orderly axiomatization of valid DL formulas, even if this means that we have to give up the finitary nature of standard axiom systems.

Below we present a complete infinitary axiomatization S2 of DL that includes an inference rule with infinitely many premises. Before doing so, however, we must get a certain technical complication out of the way. We would like to be able to consider valid first-order formulas as axiom schemes, but instantiated by general formulas of DL. In order to make formulas amenable to first-order manipulation, we must be able to make sense of such notions as "a free occurrence of x in φ" and the substitution $\varphi[x/t]$. For example, we would like to be able to use the axiom scheme of the predicate calculus $\forall x\ \varphi \rightarrow \varphi[x/t]$, even if φ contains programs.

The problem arises because the dynamic nature of the semantics of DL may cause a single occurrence of a variable in a DL formula to act as both a free and bound occurrence. For example, in the formula <**while** $x \leq 99$ **do** $x := x + 1$>**1**, the occurrence of x in the expression $x + 1$ acts as both a free occurrence (for the first assignment) and as a bound occurrence (for subsequent assignments).

There are several reasonable ways to deal with this, and we present one for definiteness. Without loss of generality, we assume that whenever required, all programs appear in the special form

$$<\overline{z} := \overline{x}\ ;\ \alpha\ ;\ \overline{x} := \overline{z}>\varphi \tag{37}$$

where $\overline{x} = (x_1, \ldots, x_n)$ and $\overline{z} = (z_1, \ldots, z_n)$ are tuples of variables, $\overline{z} := \overline{x}$ stands for

$$z_1 := x_1\ ;\ \cdots\ ;\ z_n := x_n$$

(and similarly for $\overline{x} := \overline{z}$), the x_i do not appear in α, and the z_i are new variables appearing nowhere in the relevant context outside of the program α. The idea is to make programs act on the "local" variables z_i by first copying the values of the x_i into the z_i, thus freezing the x_i, executing the program with the z_i, and then restoring the x_i. This form can be easily obtained from any DL formula by consistently changing all variables of any program to new ones and adding the appropriate assignments that copy and then restore the values. Clearly, the new formula is equivalent to the old. Given a DL formula in this form, the following are bound occurrences of variables:

- all occurrences of x in a subformula of the form $\exists x\ \varphi$;
- all occurrences of z_i in a subformula of the form (37) (note, though, that z_i does not occur in φ at all);
- all occurrences of x_i in a subformula of the form (37) except for its occurrence in the assignment $z_i := x_i$.

Every occurrence of a variable that is not bound is free. Our axiom system will have an axiom that enables free translation into the special form discussed, and in the sequel we assume that the special form is used whenever required (for example, in the assignment axiom scheme below).

As an example, consider the formula:

$$\begin{aligned}&\forall x\ (<y := f(x);\ x := g(y,x)>p(x,y)) \ \ \rightarrow \\ &\quad <z_1 := h(z);\ z_2 := y;\ z_2 := f(z_1);\ z_1 := g(z_2,z_1);\ x := z_1;\ y := z_2>p(x,y).\end{aligned}$$

Denoting $<y := f(x); x := g(y,x)>p(x,y)$ by φ, the conclusion of the implication is just $\varphi[x/h(z)]$ according to the convention above; that is, the result of replacing all free occurrences of x in φ by $h(z)$ after φ has been transformed into special form. We want the above formula to be considered a legal instance of the assignment axiom scheme below.

Axiom System S2

Axiom Schemes

- all instances of valid first-order formulas;
- all instances of valid formulas of PDL;
- $<x := t>\varphi \leftrightarrow \varphi[x/t]$;
- $\varphi \leftrightarrow \widehat{\varphi}$, where $\widehat{\varphi}$ is φ in which some occurrence of a program α has been replaced by the program $z := x;\ \alpha';\ x := z$ for z not appearing in φ, and where α' is α with all occurrences of x replaced by z.

Inference Rules

- modus ponens:

$$\frac{\varphi,\ \varphi \to \psi}{\psi}$$

- generalization:

$$\frac{\varphi}{[\alpha]\varphi} \quad \text{and} \quad \frac{\varphi}{\forall x\, \varphi}$$

- infinitary convergence:

$$\frac{\varphi \to [\alpha^n]\psi,\ n \in \omega}{\varphi \to [\alpha^*]\psi}$$

Provability in Axiom System S2, denoted by $\vdash_{S2}$, is the usual concept for systems with infinitary rules of inference; that is, deriving a formula using the infinitary rule requires infinitely many premises to have been previously derived.

Axiom System S2 consists of an axiom for assignment, facilities for propositional reasoning about programs and first-order reasoning with no programs (but with programs possibly appearing in instantiated first-order formulas), and an infinitary rule for $[\alpha^*]$. The dual construct, $<\alpha^*>$, is taken care of by the "unfolding" validity of PDL:

$$<\alpha^*>\varphi \quad \leftrightarrow \quad (\varphi \vee <\alpha;\alpha^*>\varphi).$$

THEOREM 72. *For any formula φ of* DL*,*

$$\models \varphi \quad \Leftrightarrow \quad \vdash_{S2} \varphi.$$

11.2 Interpreted Reasoning

Proving properties of real programs very often involves reasoning on the interpreted level, where one is interested in $\mathfrak{A}$-validity for a particular structure $\mathfrak{A}$. A typical proof might use induction on the length of the computation to establish an invariant for partial correctness or to exhibit a decreasing value in some well-founded set for termination. In each case, the problem is reduced to the problem of verifying some domain-dependent facts, sometimes called *verification conditions*. Mathematically speaking, this kind of activity is really an effective transformation of assertions about programs into ones about the underlying structure.

For DL, this transformation can be guided by a direct induction on program structure using an axiom system that is complete *relative to* any given arithmetical structure $\mathfrak{A}$. The essential idea is to exploit the existence, for any given DL formula, of a first-order equivalent in $\mathfrak{A}$, as guaranteed by Theorem 60. In the axiom systems we construct, instead of dealing with the Π^1_1-hardness of the validity problem by an infinitary rule, we take all $\mathfrak{A}$-valid first-order formulas as additional axioms. Relative to this set of axioms, proofs are finite and effective.

For partial correctness assertions of the form $\varphi \to [\alpha]\psi$ with φ and ψ first-order and α containing first-order tests, it suffices to show that DL reduces to the first-order logic $L_{\omega\omega}$, and there is no need for the natural numbers to be present. Thus, Axiom System S3 below works for finite structures too. Axiom System S4 is an *arithmetically complete* system for full DL that does make explicit use of natural numbers.

It follows from Theorem 66 that for partial correctness formulas we cannot hope to obtain a completeness result similar to the one proved in Theorem 71 for termination formulas. A way around this difficulty is to consider only *expressive* structures.

A structure $\mathfrak{A}$ for the first-order vocabulary Σ is said to be *expressive* for a programming language K if for every $\alpha \in K$ and for every first-order formula φ, there exists a first-order formula ψ_L such that $\mathfrak{A} \vDash \psi_L \leftrightarrow [\alpha]\varphi$. Examples of structures that are expressive for most programming languages are finite structures and arithmetical structures.

Axiom System S3

Axiom Schemes

- all instances of valid formulas of PDL;
- $<x := t>\varphi \leftrightarrow \varphi[x/t]$ for first-order φ.

Inference Rules

- modus ponens:

$$\frac{\varphi,\ \varphi \to \psi}{\psi}$$

- generalization:

$$\frac{\varphi}{[\alpha]\,\varphi}.$$

Note that Axiom System S3 is really the axiom system for PDL from Section 4 with the addition of the assignment axiom. Given a DL formula φ and a structure $\mathfrak{A}$, denote by $\mathfrak{A} \vdash_{S3} \varphi$ provability of φ in the system obtained from Axiom System S3 by adding the following set of axioms:

- all $\mathfrak{A}$-valid first-order sentences.

THEOREM 73. *For every expressive structure $\mathfrak{A}$ and for every formula ξ of* DL *of the form $\varphi \to [\alpha]\,\psi$, where φ and ψ are first-order and α involves only first-order tests, we have*

$$\mathfrak{A} \vDash \xi \quad \Leftrightarrow \quad \mathfrak{A} \vdash_{S3} \xi.$$

Now we present an axiom system S4 for full DL. It is similar in spirit to S3 in that it is complete relative to the formulas valid in the structure under consideration. However, this system works for arithmetical structures only. It is not tailored to deal with other expressive structures, notably finite ones, since it requires the use of the natural numbers. The kind of completeness result stated here is thus termed *arithmetical*.

As in Section 9.2, we state the results for the special structure $\mathbb{N}$, omitting the technicalities needed to deal with general arithmetical structures. The main difference is that in $\mathbb{N}$ we can use variables n, m, etc., knowing that their values will be natural numbers. We can thus write $n + 1$, for example, assuming the standard interpretation. When working in an unspecified arithmetical structure, we have to precede such usage with appropriate predicates that guarantee that we are indeed talking about that part of the domain that is isomorphic to the natural numbers. For example, we would often have to use the first-order formula, call it $\mathrm{nat}(n)$, which is true precisely for the elements representing natural numbers, and which exists by the definition of an arithmetical structure.

Axiom System S4

Axiom Schemes

- all instances of valid first-order formulas;
- all instances of valid formulas of PDL;
- $<x := t>\varphi \leftrightarrow \varphi[x/t]$ for first-order φ.

Inference Rules

- modus ponens:

$$\frac{\varphi,\ \varphi \to \psi}{\psi}$$

- generalization:

$$\frac{\varphi}{[\alpha]\varphi} \quad \text{and} \quad \frac{\varphi}{\forall x\, \varphi}$$

- convergence:

$$\frac{\varphi(n+1) \to <\alpha>\varphi(n)}{\varphi(n) \to <\alpha^*>\varphi(0)}$$

  for first order φ and variable n not appearing in α.

REMARK 74. For general arithmetical structures, the $+1$ and 0 in the rule of convergence denote suitable first-order definitions.

As in Axiom System S3, denote by $\mathfrak{A} \vdash_{S4} \varphi$ provability of φ in the system obtained from Axiom System S4 by adding all $\mathfrak{A}$-valid first-order sentences as axioms.

THEOREM 75. *For every formula ξ of* DL*,*

$$\mathbb{N} \vDash \xi \quad \Leftrightarrow \quad \mathbb{N} \vdash_{S4} \xi.$$

The use of the natural numbers as a device for counting down to 0 in the convergence rule of Axiom System S4 can be relaxed. In fact, any well-founded set suitably expressible in any given arithmetical structure suffices. Also, it is not necessary to require that an execution of α causes the truth of the parameterized $\varphi(n)$ in that rule to decrease exactly by 1; it suffices that the decrease is positive at each iteration.

In closing, we note that appropriately restricted versions of all axiom systems of this section are complete for $\mathsf{DL}(\mathrm{dreg})$. In particular, as pointed out in Section 2.6, the Hoare **while**-rule

$$\frac{\varphi \wedge \xi \to [\alpha]\varphi}{\varphi \to [\textbf{while}\ \xi\ \textbf{do}\ \alpha](\varphi \wedge \neg\xi)}$$

results from combining the generalization rule with the induction and test axioms of PDL, when * is restricted to appear only in the context of a **while** statement; that is, only in the form $(\xi?;p)^*;(\neg\xi)?$.

12 EXPRESSIVENESS OF DL

The subject of study in this section is the relative expressive power of languages. We will be primarily interested in comparing, on the uninterpreted level, the expressive power of various versions of DL. That is, for programming languages P_1 and P_2 we will study whether $\mathsf{DL}(P_1) \leq \mathsf{DL}(P_2)$ holds. Recall from Section 9 that the latter relation means that for each formula φ in $\mathsf{DL}(P_1)$, there is a formula ψ in $\mathsf{DL}(P_2)$ such that $\mathfrak{A}, u \vDash \varphi \leftrightarrow \psi$ for all structures $\mathfrak{A}$ and initial states u.

Studying the expressive power of logics rather than the computational power of programs allows us to compare, for example, deterministic and nondeterministic programming languages. Also, we will see that the answer to the fundamental question "$\mathsf{DL}(P_1) \leq \mathsf{DL}(P_2)$?" may depend crucially on the vocabulary over which we consider logics and programs. For this reason we always make clear in the theorems of this section our assumptions on the vocabulary.

THEOREM 76. *Let Σ be a rich vocabulary. Then*

(i) $\mathsf{DL}(\mathrm{stk}) \leq \mathsf{DL}(\mathrm{array})$.

(ii) $\mathsf{DL}(\mathrm{stk}) \equiv \mathsf{DL}(\mathrm{array})$ *iff* $P = PSPACE$.

Moreover, the same holds for deterministic regular programs with an algebraic stack and deterministic regular programs with arrays.

THEOREM 77. *Over a monadic vocabulary, nondeterministic regular programs with a Boolean stack have the same computational power as nondeterministic regular programs with an algebraic stack.*

Now we investigate the role that nondeterminism plays in the expressive power of logics of programs. As we shall see, the general conclusion is that for a programming language of sufficient computational power, nondeterminism does not increase the expressive power of the logic.

We start our discussion of the role of nondeterminism with the basic case of regular programs. Recall that DL and DDL denote the logics of nondeterministic and deterministic regular programs, respectively.

We can now state the main result that separates the expressive power of deterministic and nondeterministic **while** programs.

THEOREM 78. *For every vocabulary containing at least two unary function symbols or at least one function symbol of arity greater than one,* DDL *is strictly less expressive than* DL*; that is,* $\mathsf{DDL} < \mathsf{DL}$.

It turns out that Theorem 78 cannot be extended to vocabularies containing just one unary function symbol without solving a well known open problem in complexity theory.

THEOREM 79. *For every rich mono-unary vocabulary, the statement "*DDL *is strictly less expressive than* DL*" is equivalent to* $LOGSPACE \neq NLOGSPACE$.

We now turn our attention to the discussion of the role nondeterminism plays in the expressive power of regular programs with a Boolean stack. For a vocabulary containing at least two unary function symbols, nondeterminism increases the expressive power of DL over regular programs with a Boolean stack.

For the rest of this section, we let the vocabulary contain two unary function symbols.

THEOREM 80. *For a vocabulary containing at least two unary function symbols or a function symbol of arity greater than two,* $\mathsf{DL}(\mathrm{dbstk}) < \mathsf{DL}(\mathrm{bstk})$.

It turns out that for programming languages that use sufficiently strong data types, nondeterminism does not increase the expressive power of Dynamic Logic.

THEOREM 81. *For every vocabulary,*

(i) $\mathsf{DL}(\mathrm{dstk}) \equiv \mathsf{DL}(\mathrm{stk})$*;*

(ii) $\mathsf{DL}(\mathrm{darray}) \equiv \mathsf{DL}(\mathrm{array})$.

We will discuss the role of unbounded memory of programs for the expressive power of the corresponding logic. However, this result depends on assumptions about the vocabulary Σ.

Recall from Section 8.2 that an r.e. program α has *bounded memory* if the set $CS(\alpha)$ contains only finitely many distinct variables from V, and if in addition the nesting of function symbols in terms that occur in seqs of $CS(\alpha)$ is bounded. This restriction implies that such a program can be simulated in all interpretations by a device that uses a fixed finite number of registers, say $x_1, \ldots, x_n$, and all its elementary steps consist of either performing a test of the form

$$r(x_{i_1}, \ldots, x_{i_m})?,$$

where r is an m-ary relation symbol of Σ, or executing a simple assignment of either of the following two forms:

$$x_i := f(x_{i_1}, \ldots, x_{i_k}) \qquad\qquad x_i := x_j.$$

In general, however, such a device may need a very powerful control (that of a Turing machine) to decide which elementary step to take next.

An example of a programming language with bounded memory is the class of regular programs with a Boolean stack. Indeed, the Boolean stack strengthens the control structure of a regular program without introducing extra registers for storing algebraic elements. It can be shown without much difficulty that regular programs with a Boolean stack have bounded memory. On the other hand, regular programs with an algebraic stack or with arrays are programming languages with unbounded memory.

For monadic vocabularies, the class of nondeterministic regular programs with a Boolean stack is computationally equivalent to the class of nondeterministic regular programs with an algebraic stack. For deterministic programs, the situation is slightly different.

THEOREM 82.

(i) *For every vocabulary containing a function symbol of arity greater than one,* $\mathsf{DL}(\mathrm{dbstk}) < \mathsf{DL}(\mathrm{dstk})$ *and* $\mathsf{DL}(\mathrm{bstk}) < \mathsf{DL}(\mathrm{stk})$.

(ii) *For all monadic vocabularies,* $\mathsf{DL}(\mathrm{bstk}) \equiv \mathsf{DL}(\mathrm{stk})$.

(iii) *For all mono-unary vocabularies,* $\mathsf{DL}(\mathrm{dbstk}) \equiv \mathsf{DL}(\mathrm{dstk})$.

(iv) *For all monadic vocabularies containing at least two function symbols,* $\mathsf{DL}(\mathrm{dbstk}) < \mathsf{DL}(\mathrm{dstk})$.

Regular programs with a Boolean stack are situated between pure regular programs and regular programs with an algebraic stack. We start our discussion by comparing the expressive power of regular programs with and without a Boolean stack. The only known definite answer to this problem is given in the following result, which covers the case of deterministic programs only.

THEOREM 83.

(i) *Let the vocabulary be rich and mono-unary. Then*

$$\mathsf{DL}(\mathrm{dreg}) \equiv \mathsf{DL}(\mathrm{dstk}) \quad \Leftrightarrow \quad \mathit{LOGSPACE} = P.$$

(ii) *If the vocabulary contains at least one function symbol of arity greater than one or at least two unary function symbols, then* $\mathsf{DL}(\mathrm{dreg}) < \mathsf{DL}(\mathrm{dbstk})$.

It is not known whether Theorem 83(ii) holds for nondeterministic programs, and neither is its statement known to be equivalent to any of the well known open problems in complexity theory. In contrast, it follows from Theorems 83(i) and 82(iii) that for rich mono-unary vocabularies, $\mathsf{DL}(\mathrm{dreg}) \equiv \mathsf{DL}(\mathrm{dbstk})$ if and only if $\mathit{LOGSPACE} = P$. Hence, this problem cannot be solved without solving one of the major open problems in complexity theory.

The wildcard assignment statement $x := ?$ discussed in Section 8.2 chooses an element of the domain of computation nondeterministically and assigns it to x. It is a device that represents *unbounded nondeterminism* as opposed to the binary nondeterminism of the nondeterministic choice construct $\cup$. The programming language of regular programs augmented with wildcard assignment is not an acceptable programming language, since a wildcard assignment can produce values that are outside the substructure generated by the input.

Our first result shows that wildcard assignment increases the expressive power in quite a substantial way; it cannot be simulated even by r.e. programs.

THEOREM 84. *Let the vocabulary* Σ *contain two constants* c_1, c_2*, a binary predicate symbol* p*, the symbol* $=$ *for equality, and no other function or predicate symbols. There is a formula of* $\mathsf{DL}(\mathrm{wild})$ *that is equivalent to no formula of* $\mathsf{DL}(\mathrm{r.e.})$*, thus* $\mathsf{DL}(\mathrm{wild}) \not\leq \mathsf{DL}(\mathrm{r.e.})$.

It is not known whether any of the logics with unbounded memory are reducible to $\mathsf{DL}(\mathrm{wild})$.

When both wildcard and array assignments are allowed, it is possible to define the finiteness of (the domain of) a structure, but not in the logics with either of the additions removed. Thus, having both memory and nondeterminism unbounded provides more power than having either of them bounded.

THEOREM 85. *Let vocabulary Σ contain only the symbol of equality. There is a formula of* DL*(array+wild) equivalent to no formula of either* $\mathsf{DL}(\mathrm{array})$ *or* $\mathsf{DL}(\mathrm{wild})$.

13 VARIANTS OF DL

In this section we consider some restrictions and extensions of DL. We are interested mainly in questions of comparative expressive power on the uninterpreted level. In arithmetical structures these questions usually become trivial, since it is difficult to go beyond the power of first-order arithmetic without allowing infinitely many distinct tests in programs (see Theorems 60 and 61). In regular DL this luxury is not present.

13.1 Algorithmic Logic

Algorithmic Logic (AL) is the predecessor of Dynamic Logic. The basic system was defined by [Salwicki, 1970] and generated an extensive amount of subsequent research carried out by a group of mathematicians working in Warsaw. Two surveys of the first few years of their work can be found in [Banachowski *et al.*, 1977] and [Salwicki, 1977].

The original version of AL allowed deterministic **while** programs and formulas built from the constructs

$$\alpha\varphi \qquad \cup\,\alpha\varphi \qquad \cap\,\alpha\varphi$$

corresponding in our terminology to

$$<\alpha>\varphi \qquad <\alpha^*>\varphi \qquad \bigwedge_{n\in\omega} <\alpha^n>\varphi,$$

respectively, where α is a deterministic **while** program and φ is a quantifier-free first-order formula.

In [Mirkowska, 1980; Mirkowska, 1981a; Mirkowska, 1981b], AL was extended to allow nondeterministic **while** programs and the constructs

$$\nabla\alpha\varphi \qquad \Delta\alpha\varphi$$

corresponding in our terminology to

$$<\alpha>\varphi \qquad \mathbf{halt}(\alpha) \wedge [\alpha]\varphi \wedge <\alpha>\varphi,$$

respectively. The latter asserts that all traces of α are finite and terminate in a state satisfying φ.

A feature present in AL but not in DL is the set of "dynamic terms" in addition to dynamic formulas. For a first-order term t and a deterministic **while** program α, the meaning of the expression αt is the value of t after executing program α. If α does not halt, the meaning is undefined. Such terms can be systematically eliminated; for example, $P(x, \alpha t)$ is replaced by $\exists z\ (<\alpha>(z = t) \wedge P(x, z))$.

The emphasis in the early research on AL was in obtaining infinitary completeness results, developing normal forms for programs, investigating recursive procedures with parameters, and axiomatizing certain aspects of programming using formulas of AL. As an example of the latter, the algorithmic formula

$$(\textbf{while}\ s \neq \varepsilon\ \textbf{do}\ s := \textbf{pop}(s))\mathbf{1}$$

can be viewed as an axiom connected with the data structure *stack*. One can then investigate the consequences of such axioms within AL, regarding them as properties of the corresponding data structures.

Complete infinitary deductive systems for first-order and propositional versions are given in [Mirkowska, 1980; Mirkowska, 1981a; Mirkowska, 1981b]. The infinitary completeness results for AL are usually proved by the algebraic methods of [Rasiowa and Sikorski, 1963].

[Constable, 1977], [Constable and O'Donnell, 1978] and [Goldblatt, 1982] present logics similar to AL and DL for reasoning about deterministic **while** programs.

13.2 Well-Foundedness

As in Section 7 for PDL, we consider adding to DL assertions to the effect that programs can enter infinite computations. Here too, we shall be interested both in LDL and in RDL versions; *i.e.*, those in which **halt** α and **wf** α, respectively, have been added inductively as new formulas for any program α. As mentioned there, the connection with the more common notation **repeat** α and **loop** α (from which the L and R in the names LDL and RDL derive) is by:

$$\begin{aligned} \textbf{loop}\,\alpha &\overset{\text{def}}{\iff} \neg\textbf{halt}\,\alpha \\ \textbf{repeat}\,\alpha &\overset{\text{def}}{\iff} \neg\textbf{wf}\,\alpha. \end{aligned}$$

We now state some of the relevant results. The first concerns the addition of **halt** α:

THEOREM 86. LDL $\equiv$ DL.

In contrast to this, we have:

THEOREM 87. LDL $<$ RDL.

Turning to the validity problem for these extensions, clearly they cannot be any harder to decide than that of DL, which is Π^1_1-complete. However, the following result shows that detecting the absence of infinite computations of even simple uninterpreted programs is extremely hard.

THEOREM 88. *The validity problems for formulas of the form $\varphi \rightarrow$ **wf** α and formulas of the form $\varphi \rightarrow$ **halt** α, for first-order φ and regular α, are both Π^1_1-complete. If α is constrained to have only first-order tests then the $\varphi \rightarrow$ **wf** α case remains Π^1_1-complete but the $\varphi \rightarrow$ **halt** α case is r.e.; that is, it is Σ^0_1-complete.*

We just mention here that the additions to Axiom System S4 of Section 11 that are used to obtain an arithmetically complete system for RDL are the axiom

$$[\alpha^*](\varphi \to \langle\alpha\rangle\varphi) \quad \to \quad (\varphi \to \neg \mathbf{wf}\,\alpha)$$

and the inference rule

$$\frac{\varphi(n+1) \to [\alpha]\varphi(n),\ \neg\varphi(0)}{\varphi(n) \to \mathbf{wf}\,\alpha}$$

for first-order φ and n not occurring in α.

13.3 Probabilistic Programs

There is wide interest recently in programs that employ probabilistic moves such as coin tossing or random number draws and whose behavior is described probabilistically (for example, α is "correct" if it does what it is meant to do with probability 1). To give one well known example taken from [Miller, 1976] and [Rabin, 1980], there are fast probabilistic algorithms for checking primality of numbers but no known fast nonprobabilistic ones. Many synchronization problems including digital contract signing, guaranteeing mutual exclusion, etc. are often solved by probabilistic means.

This interest has prompted research into formal and informal methods for reasoning about probabilistic programs. It should be noted that such methods are also applicable for reasoning probabilistically about ordinary programs, for example, in average-case complexity analysis of a program, where inputs are regarded as coming from some set with a probability distribution.

[Kozen, 1981d] provided a formal semantics for probabilistic first-order **while** programs with a random assignment statement $x := ?$. Here the term "random" is quite appropriate (contrast with Section 8.2) as the statement essentially picks an element out of some fixed distribution over the domain D. This domain is assumed to be given with an appropriate set of measurable subsets. Programs are then interpreted as measurable functions on a certain measurable product space of copies of D.

In [Feldman and Harel, 1984] a probabilistic version of first-order Dynamic Logic, $Pr(\mathsf{DL})$, was investigated on the interpreted level. Kozen's semantics is extended as described below to a semantics for formulas that are closed under Boolean connectives and quantification over reals and integers and that employ terms of the form $Fr(\varphi)$ for first-order φ. In addition, if α is a **while** program with nondeterministic assignments and φ is a formula, then $\{\alpha\}\varphi$ is a new formula.

The semantics assumes a domain D, say the reals, with a measure space consisting of an appropriate family of *measurable subsets* of D. The states $\mu, \nu, \ldots$ are then taken to be the positive measures on this measure space. Terms are interpreted as functions from states to real numbers, with $Fr(\varphi)$ in μ being the *frequency* (or simply, the *measure*) of φ in μ. Frequency is to positive measures as probability is to probability measures. The formula $\{\alpha\}\varphi$ is true in μ if φ is true in ν, the state (*i.e.*, measure) that is the result of applying α to μ in Kozen's semantics. Thus $\{\alpha\}\varphi$ means "after α, φ" and is the construct analogous to $<\alpha>\varphi$ of DL.

For example, in $Pr(\mathsf{DL})$ one can write

$$Fr(1) = 1 \quad \rightarrow \quad \{\alpha\}Fr(1) \geq p$$

to mean, "α halts with probability at least p." The formula

$$\begin{aligned} Fr(1) = 1 \quad \rightarrow \quad & [i := 1; x := ?; \textbf{while } x > 1/2 \textbf{ do } (x := ?; i := i + 1)] \\ & \forall n\, ((n \geq 1 \rightarrow Fr(i = n) = 2^{-n}) \wedge (n < 1 \rightarrow Fr(i = n) = 0)) \end{aligned}$$

is valid in all structures in which the distribution of the random variable used in $x := ?$ is a uniform distribution on the real interval $[0, 1]$.

An axiom system for $Pr(\mathsf{DL})$ was proved in [Feldman and Harel, 1984] to be complete relative to an extension of first-order analysis with integer variables, and for discrete probabilities first-order analysis with integer variables was shown to suffice.

14 OTHER APPROACHES

Here we discuss briefly some topics closely related to Dynamic Logic.

14.1 Logic of Effective Definitions

The Logic of Effective Definitions (LED), introduced by [Tiuryn, 1981a], was intended to study notions of computability over abtract models and to provide a universal framework for the study of logics of programs over such models. It consists of first-order logic augmented with new atomic formulas of the form $\alpha = \beta$, where α and β are *effective definitional schemes* (the latter notion is due to [Friedman, 1971]):

if φ_1 **then** t_1
  else if φ_2 **then** t_2
    else if φ_3 **then** t_3
      else if ...

where the φ_i are quantifier-free formulas and t_i are terms over a bounded set of variables, and the function $i \mapsto (\varphi_i, t_i)$ is recursive. The formula $\alpha = \beta$ is defined to be true in a state if both α and β terminate and yield the same value, or neither terminates.

Model theory and infinitary completeness of LED are treated in [Tiuryn, 1981a].

Effective definitional schemes in the definition of LED can be replaced by any programming language K, giving rise to various logical formalisms. The following result, which relates LED to other logics discussed here, is proved in [Meyer and Tiuryn, 1981; Meyer and Tiuryn, 1984].

THEOREM 89. *For every vocabulary L,* $\mathsf{LED} \equiv \mathsf{DL}(\text{r.e.})$.

14.2 Temporal Logic

Temporal Logic (TL) is an alternative application of modal logic to program specification and verification. It was first proposed as a useful tool in program verification by [Pnueli, 1977] and has since been developed by many authors in various forms. This topic is surveyed in depth in [Emerson, 1990] and [Gabbay *et al.*, 1994].

TL differs from DL chiefly in that it is *endogenous*; that is, programs are not explicit in the language. Every application has a single program associated with it, and the language may contain program-specific statements such as $\mathbf{at}\ L$, meaning "execution is currently at location L in the program." There are two competing semantics, giving rise to two different theories called *linear-time* and *branching-time* TL. In the former, a model is a linear sequence of program states representing an execution sequence of a deterministic program or a possible execution sequence of a nondeterministic or concurrent program. In the latter, a model is a tree of program states representing the space of all possible traces of a nondeterministic or concurrent program. Depending on the application and the semantics, different syntactic constructs can be chosen. The relative advantages of linear and branching time semantics are discussed in [Lamport, 1980; Emerson and Halpern, 1986; Emerson and Lei, 1987; Vardi, 1998a].

Modal constructs used in TL include

$\Box\varphi$	"φ holds in all future states"
$\Diamond\varphi$	"φ holds in some future state"
$\bigcirc\varphi$	"φ holds in the next state"
$\varphi\,\mathbf{until}\,\psi$	"there exists some strictly future point t at which ψ will be satisfied and all points strictly between the current state and t satisfy φ"

for linear-time logic, as well as constructs for expressing

"for all traces starting from the present state ... "
"for some trace starting from the present state ... "

for branching-time logic.

Temporal logic is useful in situations where programs are not normally supposed to halt, such as operating systems, and is particularly well suited to the study of concurrency. Many classical program verification methods such as the *intermittent assertions method* are treated quite elegantly in this framework.

Temporal logic has been most successful in providing tools for proving properties of concurrent *finite state* protocols, such as solutions to the *dining philosophers* and *mutual exclusion* problems, which are popular abstract versions of synchronization and resource management problems in distributed systems.

The induction principle of TL takes the form:

$$\varphi \wedge \Box(\varphi \to \bigcirc\varphi) \quad \to \quad \Box\varphi. \tag{38}$$

Note the similarity to the PDL induction axiom (Axiom 17(viii)):

$$\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi) \quad \to \quad [\alpha^*]\varphi.$$

This is a classical program verification method known as *inductive* or *invariant assertions*.

The operators $\bigcirc$, $\Diamond$, and $\Box$ can all be defined in terms of $\mathbf{until}$:

$$\begin{array}{lcl} \bigcirc\varphi & \Leftrightarrow & \neg(\mathbf{0}\,\mathbf{until}\,\neg\varphi) \\ \Diamond\varphi & \Leftrightarrow & \varphi \vee (\mathbf{1}\,\mathbf{until}\,\varphi) \\ \Box\varphi & \Leftrightarrow & \varphi \wedge \neg(\mathbf{1}\,\mathbf{until}\,\neg\varphi), \end{array}$$

but not vice-versa. It has been shown in [Kamp, 1968] and [Gabbay *et al.*, 1980] that the $\mathbf{until}$ operator is powerful enough to express anything that can be expressed in the first-order theory of $(\omega, <)$. It has also been shown in [Wolper, 1981;

Wolper, 1983] that there are very simple predicates that cannot be expressed by **until**; for example, "φ is true at every multiple of 4."

The **until** operator has been shown to be very useful in expressing properties of programs that are not properties of the input/output relation, such as: "If process p requests a resource before q does, then it will receive it before q does." Indeed, much of the research in TL has concentrated on providing useful methods for proving these and other kinds of properties (see [Manna and Pnueli, 1981; Gabbay *et al.*, 1980]).

Concurrency and Nondeterminism

Unlike DL, TL can be applied to programs that are not normally supposed to halt, such as operating systems, because programs are interpreted as *traces* instead of pairs of states.

Up to now we have only considered deterministic, single-process programs. There is no reason however not to apply TL to *nondeterministic* and *concurrent (multiprocessor)* systems, in which next states are not unique. The computation is no longer a single trace, but many different traces are possible. We can assemble them all together to get a *computation tree* in which each node represents a state accessible from the start state.

As above, an *invariance property* is a property of the form $\Box\varphi$. However, the dual $\Diamond$ of the operator $\Box$ defined in this way does not really capture what we mean by *eventuality* or *liveness* properties. We would like to be able to say that *every* possible trace in the computation tree has a state satisfying φ. For instance, a nondeterministic program is *total* if there is no chance of an infinite trace out of the start state s; that is, every trace out of s satisfies $\Diamond$**halt**. The dual $\Diamond$ of $\Box$ as defined by $\Diamond\varphi = \neg\Box\neg\varphi$ does not really express this. It says instead

$$s \vDash \Diamond\varphi \quad \Leftrightarrow \quad \text{there is } \textit{some} \text{ node } t \text{ in the tree below } s \text{ such that } t \vDash \varphi.$$

This is not a very useful statement.

One way to fix this is to introduce the branching time operator A that says, "For all traces in the tree . . . ," and then use $\Box$, $\Diamond$ in the sense of linear TL applied to the trace quantified by A. The dual of A is E, which says, "There exists a trace in the tree . . . ." Thus, in order to say that the computation tree starting from the current state satisfies a safety or invariance property, we would write

$$\mathsf{A}\Box\varphi,$$

which says, "For all traces π out of the current state, π satisfies $\Box\varphi$," and to say that the tree satisfies an eventuality property, we would write

$$\mathsf{A}\Diamond\varphi,$$

which says, "For all traces π out of the current state, π satisfies $\Diamond\varphi$; that is, φ occurs somewhere along the trace π." The logic with the linear temporal operators augmented with the trace quantifiers A and E is known as CTL; see [Emerson, 1990; Emerson and Halpern, 1986; Emerson and Halpern, 1985; Emerson and Lei, 1987; Emerson and Sistla, 1984].

Complexity and Deductive Completeness

A useful axiomatization of linear-time TL without the until operator is given by the axioms

$$\begin{array}{rcl}
\Box(\varphi \to \psi) & \to & (\Box\varphi \to \Box\psi) \\
\Box(\varphi \wedge \psi) & \leftrightarrow & \Box\varphi \wedge \Box\psi \\
\Diamond\varphi & \leftrightarrow & \varphi \vee \bigcirc\Diamond\varphi \\
\bigcirc(\varphi \vee \psi) & \leftrightarrow & \bigcirc\varphi \vee \bigcirc\psi \\
\bigcirc(\varphi \wedge \psi) & \leftrightarrow & \bigcirc\varphi \wedge \bigcirc\psi \\
\varphi \wedge \Box(\varphi \to \bigcirc\varphi) & \to & \Box\varphi \\
\forall x\, \varphi(x) & \to & \varphi(t) \;\; (t \text{ is free for } x \text{ in } \varphi) \\
\forall x\, \Box\varphi & \to & \Box\forall x\, \varphi
\end{array}$$

and rules

$$\frac{\varphi, \quad \varphi \to \psi}{\psi} \qquad \frac{\varphi}{\Box\varphi} \qquad \frac{\varphi}{\forall x\, \varphi}.$$

Compare the axioms of PDL (Axioms 17). The propositional fragment of this deductive system is complete for linear-time propositional TL, as shown in [Gabbay *et al.*, 1980].

[Sistla and Clarke, 1982] and [Emerson and Halpern, 1985] have shown that the validity problem for most versions of propositional TL is *PSPACE*-complete for linear structures and *EXPTIME*-complete for branching structures.

Embedding TL *in* DL

TL is subsumed by DL. To embed propositional TL into PDL, take an atomic program a to mean "one step of program p." In the linear model, the TL constructs $\bigcirc\varphi$, $\Box\varphi$, $\Diamond\varphi$, and $\varphi\,\mathbf{until}\,\psi$ are then expressed by $[a]\varphi$, $[a^*]\varphi$, $<a^*>\varphi$, and $<(a;\varphi?)^*;a>\psi$, respectively.

14.3 Process Logic

Dynamic Logic and Temporal Logic embody markedly different approaches to reasoning about programs. This dichotomy has prompted researchers to search for an appropriate process logic that combines the best features of both. An appropriate candidate should combine the ability to reason about programs compositionally with the ability to reason directly about the intermediate states encountered during the course of a computation.

[Pratt, 1979c], [Parikh, 1978b], [Nishimura, 1980], and [Harel *et al.*, 1982b] all suggested increasingly more powerful propositional-level formalisms in which the basic idea is to interpret formulas in *traces* rather than in states. In particular, [Harel *et al.*, 1982b] present a system called *Process Logic* (PL), which is essentially a union of TL and test-free regular PDL. That paper proves that the satisfiability problem is decidable and gives a complete finitary axiomatization.

Syntactically, we have programs $\alpha, \beta, \ldots$ and propositions $\varphi, \psi, \ldots$ as in PDL. We have atomic symbols of each type and compound expressions built up from the operators $\rightarrow$, $\mathbf{0}$, ;, $\cup$, *, ? (applied to Boolean combinations of atomic formulas only), ω, and $[\,]$. In addition we have the temporal operators $\mathbf{first}$ and $\mathbf{until}$. The temporal operators are available for expressing and reasoning about trace properties, but programs are constructed compositionally as in PDL. Other operators are defined as in PDL (see Section 2.1) except for **skip**, which is handled specially.

Semantically, both programs and propositions are interpreted as sets of traces. We start with a Kripke frame $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ as in Section 2.2, where K is a set of *states* $s, t, \ldots$ and the function $\mathfrak{m}_{\mathfrak{K}}$ interprets atomic formulas p as subsets of K and atomic programs a as binary relations on K. The temporal operators are defined as in TL.

Trace models satisfy (most of) the PDL axioms. As in Section 14.2, define

$$\begin{array}{rcl} \mathbf{halt} & \stackrel{\text{def}}{\Longleftrightarrow} & \bigcirc\mathbf{0} \\ \mathbf{fin} & \stackrel{\text{def}}{\Longleftrightarrow} & \Diamond\mathbf{halt} \\ \mathbf{inf} & \stackrel{\text{def}}{\Longleftrightarrow} & \neg\mathbf{fin}, \end{array}$$

which say that the trace is of length 0, of finite length, or of infinite length, respectively. Define two new operators $[\![\ ]\!]$ and $\ll\ \gg$:

$$\begin{array}{rcl} [\![\alpha]\!]\varphi & \stackrel{\text{def}}{\Longleftrightarrow} & \mathbf{fin} \rightarrow [\alpha]\varphi \\ \ll\!\alpha\!\gg\varphi & \stackrel{\text{def}}{\Longleftrightarrow} & \neg[\![\alpha]\!]\neg\varphi \;\Leftrightarrow\; \mathbf{fin} \wedge \langle\alpha\rangle\varphi. \end{array}$$

The * operator is the same as in PDL. It can be shown that the two PDL axioms

$$\begin{array}{rcl} \varphi \wedge [\alpha][\alpha^*]\varphi & \leftrightarrow & [\alpha^*]\varphi \\ \varphi \wedge [\alpha^*](\varphi \rightarrow [\alpha]\varphi) & \rightarrow & [\alpha^*]\varphi \end{array}$$

hold by establishing that

$$\begin{array}{rcl} \bigcup_{n\geq 0} \mathfrak{m}_{\mathfrak{K}}(\alpha^n) & = & \mathfrak{m}_{\mathfrak{K}}(\alpha^0) \cup (\mathfrak{m}_{\mathfrak{K}}(\alpha) \circ \bigcup_{n\geq 0} \mathfrak{m}_{\mathfrak{K}}(\alpha^n)) \\ & = & \mathfrak{m}_{\mathfrak{K}}(\alpha^0) \cup ((\bigcup_{n\geq 0} \mathfrak{m}_{\mathfrak{K}}(\alpha^n)) \circ \mathfrak{m}_{\mathfrak{K}}(\alpha)). \end{array}$$

As mentioned, the version of PL of [Harel *et al.*, 1982b] is decidable (but, it seems, in nonelementary time only) and complete. It has also been shown that if we restrict the semantics to include only finite traces (not a necessary restriction for obtaining the results above), then PL is no more expressive than PDL. Translations of PL structures into PDL structures have also been investigated, making possible an elementary time decision procedure for deterministic PL; see [Halpern, 1982; Halpern, 1983]. An extension of PL in which **first** and **until** are replaced by regular operators on formulas has been shown to be decidable but nonelementary in [Harel *et al.*, 1982b]. This logic perhaps comes closer to the desired objective of a powerful decidable logic of traces with natural syntactic operators that is closed under attachment of regular programs to formulas.

14.4 The μ-Calculus

The μ-calculus was suggested as a formalism for reasoning about programs in [Scott and de Bakker, 1969] and was further developed in [Hitchcock and Park, 1972], [Park, 1976], and [de Bakker, 1980].

The heart of the approach is μ, the *least fixpoint* operator, which captures the notions of iteration and recursion. The calculus was originally defined as a first-order-level formalism, but propositional versions have become popular.

The μ operator binds relation variables. If $\varphi(X)$ is a logical expression with a free relation variable X, then the expression $\mu X.\varphi(X)$represents the least X such that $\varphi(X) = X$, if such an X exists. For example, the reflexive transitive closure R^* of a binary relation R is the least binary relation containing R and closed under reflexivity and transitivity; this would be expressed in the first-order μ-calculus as

$$R^* \stackrel{\text{def}}{=} \mu X(x,y).(x = y \vee \exists z\ (R(x,z) \wedge X(z,y))). \tag{39}$$

This should be read as, "the least binary relation $X(x,y)$ such that either $x = y$ or x is related by R to some z such that z and y are already related by X." This captures the usual fixpoint formulation of reflexive transitive closure. The formula (39) can be regarded either as a recursive program computing R^* or as an inductively defined assertion that is true of a pair (x,y) iff that pair is in the reflexive transitive closure of R.

The existence of a least fixpoint is not guaranteed except under certain restrictions. Indeed, the formula $\neg X$ has no fixpoint, therefore $\mu X.\neg X$ does not exist. Typically, one restricts the application of the binding operator μX to formulas that are *positive* or *syntactically monotone* in X; that is, those formulas in which every free occurrence of X occurs in the scope of an even number of negations. This implies that the relation operator $X \mapsto \varphi(X)$ is (semantically) monotone, which by the Knaster–Tarski theorem ensures the existence of a least fixpoint.

The first-order μ-calculus can define all sets definable by first-order induction and more. In particular, it can capture the input/output relation of any program built from any of the DL programming constructs we have discussed. Since the first-order μ-calculus also admits first-order quantification, it is easily seen to be as powerful as DL.

It was shown by [Park, 1976] that finiteness is not definable in the first-order μ-calculus with the monotonicity restriction, but well-foundedness is. Thus this version of the μ-calculus is independent of $L_{\omega_1^{\mathrm{ck}}\omega}$ (and hence of $\mathsf{DL}(\mathrm{r.e.})$) in expressive power. Well-foundedness of a binary relation R can be written

$$\forall x\ (\mu X(x).\forall y\ (R(y,x) \to X(y))).$$

A more severe syntactic restriction on the binding operator μX is to allow its application only to formulas that are *syntactically continuous* in X; that is, those formulas in which X does not occur free in the scope of any negation or any universal quantifier. It can be shown that this syntactic restriction implies semantic continuity, so the least fixpoint is the union of $\varnothing$, $\varphi(\varnothing)$, $\varphi(\varphi(\varnothing)), \ldots$. As shown in [Park, 1976], this version is strictly weaker than $L_{\omega_1^{\mathrm{ck}}\omega}$.

In [Pratt, 1981a] and [Kozen, 1982; Kozen, 1983], propositional versions of the μ-calculus were introduced. The latter version consists of propositional modal logic with a least fixpoint operator. It is the most powerful logic of its type, subsuming all known variants of PDL, game logic of [Parikh, 1983], various forms of temporal logic (see Section 14.2), and other seemingly stronger forms of the μ-calculus ([Vardi and Wolper, 1986b]). In the following presentation we focus on this version, since it has gained fairly widespread acceptance; see [Kozen, 1984; Kozen and Parikh, 1983; Streett, 1985b; Streett and Emerson, 1984; Vardi and Wolper, 1986b; Walukiewicz, 1993; Walukiewicz, 1995; Walukiewicz, 2000; Stirling, 1992; Mader, 1997; Kaivola, 1997].

The language of the propositional μ-calculus, also called the *modal μ-calculus*, is syntactically simpler than PDL. It consists of the usual propositional constructs $\to$ and **0**, atomic modalities $[a]$, and the least fixpoint operator μ. A greatest fixpoint operator dual to μ can be defined:

$$\nu X.\varphi(X) \quad \stackrel{\mathrm{def}}{\iff} \quad \neg\mu X.\neg\varphi(\neg X).$$

Variables are monadic, and the μ operator may be applied only to syntactically monotone formulas. As discussed above, this ensures monotonicity of the corresponding set operator. The language is interpreted over Kripke frames in which atomic propositions are interpreted as sets of states and atomic programs are interpreted as binary relations on states.

The propositional μ-calculus subsumes PDL. For example, the PDL formula $<a^*>\varphi$ for atomic a can be written $\mu X.(\varphi \vee <a>X)$. The formula $\mu X.<a>[a]X$, which expresses the existence of a forced win for the first player in a two-player game, and the formula $\mu X.[a]X$, which expresses well-foundedness and is equivalent to **wf** a (see Section 7), are both inexpressible in PDL, as shown in [Streett, 1981; Kozen, 1981c]. [Niwinski, 1984] has shown that even with the addition of the **halt** construct, PDL is strictly less expressive than the μ-calculus.

The propositional μ-calculus satisfies a finite model theorem, as first shown in [Kozen, 1988]. Progressively better decidability results were obtained in [Kozen and Parikh, 1983; Vardi and Stockmeyer, 1985; Vardi, 1985b], culminating in a deterministic exponential-time algorithm of [Emerson and Jutla, 1988] based on an automata-theoretic lemma of [Safra, 1988]. Since the μ-calculus subsumes PDL, it is *EXPTIME*-complete.

In [Kozen, 1982; Kozen, 1983], an axiomatization of the propositional μ-calculus was proposed and conjectured to be complete. The axiomatization consists of the axioms and rules of propositional modal logic, plus the axiom

$$\varphi[X/\mu X.\varphi] \quad \rightarrow \quad \mu X.\varphi$$

and rule

$$\frac{\varphi[X/\psi] \;\rightarrow\; \psi}{\mu X.\varphi \;\rightarrow\; \psi}$$

for μ. Completeness of this deductive system for a syntactically restricted subset of formulas was shown in [Kozen, 1982; Kozen, 1983]. Completeness for the full language was proved by [Walukiewicz, 1995; Walukiewicz, 2000]. This was quickly followed by simpler alternative proofs by [Ambler *et al.*, 1995; Bonsangue and Kwiatkowska, 1995; Hartonas, 1998]. [Bradfield, 1996] showed that the alternating μ/ν hierarchy (least/greatest fixpoints) is strict. An interesting open question is the complexity of *model checking*: does a given formula of the propositional μ-calculus hold in a given state of a given Kripke frame? Although some progress has been made (see [Bhat and Cleaveland, 1996; Cleaveland, 1996; Emerson and Lei, 1986; Sokolsky and Smolka, 1994; Stirling and Walker, 1989]), it is still unknown whether this problem has a polynomial-time algorithm.

The propositional μ-calculus has become a popular system for the specification and verification of properties of transition systems, where it has had some practical impact ([Steffen *et al.*, 1996]). Several recent papers on model checking work in this context; see [Bhat and Cleaveland, 1996; Cleaveland, 1996; Emerson and Lei, 1986; Sokolsky and Smolka, 1994; Stirling and Walker, 1989]. A comprehensive introduction can be found in [Stirling, 1992].

14.5 Kleene Algebra

Kleene algebra (KA) is the algebra of regular expressions. It is named for the mathematician S. C. Kleene (1909–1994), who among his many other achievements invented regular expressions and proved their equivalence to finite automata in [Kleene, 1956].

Kleene algebra has appeared in various guises and under many names in relational algebra ([Ng, 1984; Ng and Tarski, 1977]), semantics and logics of programs ([Kozen, 1981b; Pratt, 1988]), automata and formal language theory ([Kuich, 1987; Kuich and Salomaa, 1986]), and the design and analysis of algorithms ([Aho *et al.*, 1975; Tarjan, 1981; Mehlhorn, 1984; Iwano and Steiglitz, 1990; Kozen, 1991b]). As discussed in Section 13, Kleene algebra plays a prominent role in dynamic algebra as an algebraic model of program behavior.

Beginning with the monograph of [Conway, 1971], many authors have contributed over the years to the development of the algebraic theory; see [Backhouse, 1975; Krob, 1991; Kleene, 1956; Kuich and Salomaa, 1986; Sakarovitch, 1987; Kozen, 1990; Bloom and Ésik, 1992; Hopkins and Kozen, 1999]. See also [Kozen, 1996] for further references.

A *Kleene algebra* is an algebraic structure $(K,\ +,\ \cdot,\ ^*,\ 0,\ 1)$ satisfying the axioms

$$\begin{aligned}
\alpha + (\beta + \gamma) &= (\alpha + \beta) + \gamma \\
\alpha + \beta &= \beta + \alpha \\
\alpha + 0 &= \alpha + \alpha \ = \ \alpha \\
\alpha(\beta\gamma) &= (\alpha\beta)\gamma \\
1\alpha &= \alpha 1 \ = \ \alpha \\
\alpha(\beta + \gamma) &= \alpha\beta + \alpha\gamma \\
(\alpha + \beta)\gamma &= \alpha\gamma + \beta\gamma \\
0\alpha &= \alpha 0 \ = \ 0 \\
1 + \alpha\alpha^* &= 1 + \alpha^*\alpha \ = \ \alpha^* && (40) \\
\beta + \alpha\gamma \leq \gamma &\rightarrow \alpha^*\beta \leq \gamma && (41) \\
\beta + \gamma\alpha \leq \gamma &\rightarrow \beta\alpha^* \leq \gamma && (42)
\end{aligned}$$

where $\leq$ refers to the natural partial order on K:

$$\alpha \leq \beta \quad \stackrel{\text{def}}{\iff} \quad \alpha + \beta = \beta.$$

In short, a KA is an idempotent semiring under $+$, $\cdot$, 0, 1 such that $\alpha^*\beta$ is the least solution to $\beta + \alpha x \leq x$ and $\beta\alpha^*$ is the least solution to $\beta + x\alpha \leq x$. The axioms (40)–(42) say essentially that * behaves like the asterate operator on sets of strings or reflexive transitive closure on binary relations. This particular axiomatization is from [Kozen, 1991a; Kozen, 1994a], but there are other competing ones.

The axioms (41) and (42) correspond to the reflexive transitive closure rule (RTC) of PDL (Section 2.5). Instead, we might postulate the equivalent axioms

$$\begin{aligned}
\alpha\gamma \leq \gamma &\rightarrow \alpha^*\gamma \leq \gamma && (43) \\
\gamma\alpha \leq \gamma &\rightarrow \gamma\alpha^* \leq \gamma, && (44)
\end{aligned}$$

which correspond to the loop invariance rule (LI). The induction axiom (IND) is inexpressible in KA, since there is no negation.

A Kleene algebra is *-*continuous* if it satisfies the infinitary condition

$$\alpha\beta^*\gamma \quad = \quad \sup_{n \geq 0} \alpha\beta^n\gamma \qquad (45)$$

where

$$\beta^0 \stackrel{\text{def}}{=} 1 \qquad \beta^{n+1} \stackrel{\text{def}}{=} \beta\beta^n$$

and where the supremum is with respect to the natural order $\leq$. We can think of (45) as a conjunction of the infinitely many axioms $\alpha\beta^n\gamma \leq \alpha\beta^*\gamma$, $n \geq 0$, and the infinitary Horn formula

$$(\bigwedge_{n\geq 0} \alpha\beta^n\gamma \leq \delta) \quad \rightarrow \quad \alpha\beta^*\gamma \leq \delta.$$

In the presence of the other axioms, the *-continuity condition (45) implies (41)–(44) and is strictly stronger in the sense that there exist Kleene algebras that are not *-continuous ([Kozen, 1990]).

The fundamental motivating example of a Kleene algebra is the family of regular sets of strings over a finite alphabet, but other classes of structures share the same equational theory, notably the binary relations on a set. In fact it is the latter interpretation that makes Kleene algebra a suitable choice for modeling programs in dynamic algebras. Other more unusual interpretations are the $\min, +$ algebra used in shortest path algorithms (see [Aho *et al.*, 1975; Tarjan, 1981; Mehlhorn, 1984; Kozen, 1991b]) and KAs of convex polyhedra used in computational geometry as described in [Iwano and Steiglitz, 1990].

Axiomatization of the equational theory of the regular sets is a central question going back to the original paper of [Kleene, 1956]. A completeness theorem for relational algebras was given in an extended language by [Ng, 1984; Ng and Tarski, 1977]. Axiomatization is a central focus of the monograph of [Conway, 1971], but the bulk of his treatment is infinitary. [Redko, 1964] proved that there is no finite equational axiomatization. Schematic equational axiomatizations for the algebra of regular sets, necessarily representing infinitely many equations, have been given by [Krob, 1991] and [Bloom and Ésik, 1993]. [Salomaa, 1966] gave two finitary complete axiomatizations that are sound for the regular sets but not sound in general over other standard interpretations, including relational interpretations. The axiomatization given above is a finitary universal Horn axiomatization that is sound and complete for the equational theory of standard relational and language-theoretic models, including the regular sets ([Kozen, 1991a; Kozen, 1994a]). Other work on completeness appears in [Krob, 1991; Boffa, 1990; Boffa, 1995; Archangelsky, 1992].

The literature contains a bewildering array of inequivalent definitions of Kleene algebras and related algebraic structures; see [Conway, 1971; Pratt, 1988; Pratt, 1990; Kozen, 1981b; Kozen, 1991a; Aho *et al.*, 1975; Mehlhorn, 1984; Kuich, 1987; Kozen, 1994b]. As demonstrated in [Kozen, 1990], many of these are strongly related. One important property shared by most of them is closure under the formation of $n \times n$ matrices. This was proved for the axiomatization above in [Kozen, 1991a; Kozen, 1994a], but the idea essentially goes back to [Kleene, 1956; Conway, 1971; Backhouse, 1975]. This result gives rise to an algebraic treatment of finite automata in which the automata are represented by their transition matrices.

The equational theory of Kleene algebra is *PSPACE*-complete ([Stockmeyer and Meyer, 1973]); thus it is apparently less complex than PDL, which is *EXPTIME*-complete (Theorem 21), although the strict separation of the two complexity classes is still open.

Kleene Algebra with Tests

From a practical standpoint, many simple program manipulations such as loop unwinding and basic safety analysis do not require the full power of PDL, but can be carried out in a purely equational subsystem using the axioms of Kleene algebra. However, *tests* are an essential ingredient, since they are needed to model conventional programming constructs such as conditionals and **while** loops and to handle assertions. This motivates the definition of the following variant of KA introduced in [Kozen, 1996; Kozen, 1997b].

A *Kleene algebra with tests* (KAT) is a Kleene algebra with an embedded Boolean subalgebra. Formally, it is a two-sorted algebra

$$(K,\ B,\ +,\ \cdot,\ {}^*,\ {}^-,\ 0,\ 1)$$

such that

- $(K,\ +,\ \cdot,\ {}^*,\ 0,\ 1)$ is a Kleene algebra
- $(B,\ +,\ \cdot,\ {}^-,\ 0,\ 1)$ is a Boolean algebra
- $B\ \subseteq\ K$.

The unary negation operator $^-$ is defined only on B. Elements of B are called *tests* and are written $\varphi, \psi, \ldots$. Elements of K (including elements of B) are written $\alpha, \beta, \ldots$. In PDL, a test would be written φ?, but in KAT we dispense with the symbol ?.

This deceptively concise definition actually carries a lot of information. The operators $+, \cdot, 0, 1$ each play two roles: applied to arbitrary elements of K, they refer to nondeterministic choice, composition, fail, and skip, respectively; and applied to tests, they take on the additional meaning of Boolean disjunction, conjunction, falsity, and truth, respectively. These two usages do not conflict—for example, sequential testing of two tests is the same as testing their conjunction—and their coexistence admits considerable economy of expression.

For applications in program verification, the standard interpretation would be a Kleene algebra of binary relations on a set and the Boolean algebra of subsets of the identity relation. One could also consider trace models, in which the Kleene elements are sets of traces (sequences of states) and the Boolean elements are sets of states (traces of length 0). As with KA, one can form the algebra $n \times$

n matrices over a KAT $(K,\ B)$; the Boolean elements of this structure are the diagonal matrices over B.

KAT can express conventional imperative programming constructs such as conditionals and while loops as in PDL. It can perform elementary program manipulation such as loop unwinding, constant propagation, and basic safety analysis in a purely equational manner. The applicability of KAT and related equational systems in practical program verification has been explored in [Cohen, 1994a; Cohen, 1994b; Cohen, 1994c; Kozen, 1996; Kozen and Patron, 2000].

There is a language-theoretic model that plays the same role in KAT that the regular sets play in KA, namely the algebra of regular sets of *guarded strings*, and a corresponding completeness result was obtained by [Kozen and Smith, 1996]. Moreover, KAT is complete for the equational theory of relational models, as shown in [Kozen and Smith, 1996]. Although less expressive than PDL, KAT is also apparently less difficult to decide: it is *PSPACE*-complete, the same as KA, as shown in [Cohen *et al.*, 1996].

In [Kozen, 1999a], it is shown that KAT subsumes propositional Hoare Logic in the following sense. The partial correctness assertion $\{\varphi\}\ \alpha\ \{\psi\}$ is encoded in KAT as the equation $\varphi\alpha\overline{\psi} = 0$, or equivalently $\varphi\alpha = \varphi\alpha\psi$. If a rule

$$\frac{\{\varphi_1\}\ \alpha_1\ \{\psi_1\},\ \ldots,\ \{\varphi_n\}\ \alpha_n\ \{\psi_n\}}{\{\varphi\}\ \alpha\ \{\psi\}}$$

is derivable in propositional Hoare Logic, then its translation, the universal Horn formula

$$\varphi_1\alpha_1\overline{\psi}_1 = 0 \land \cdots \land \varphi_n\alpha_n\overline{\psi}_n = 0 \quad \to \quad \varphi\alpha\overline{\psi} = 0,$$

is a theorem of KAT. For example, the **while** rule of Hoare logic (see Section 2.6) becomes

$$\sigma\varphi\alpha\overline{\varphi} = 0 \quad \to \quad \varphi(\sigma\alpha)^*\overline{\sigma}\,\overline{\overline{\sigma}\varphi} = 0.$$

More generally, all relationally valid Horn formulas of the form

$$\gamma_1 = 0 \land \cdots \land \gamma_n = 0 \quad \to \quad \alpha = \beta$$

are theorems of KAT ([Kozen, 1999a]).

Horn formulas are important from a practical standpoint. For example, commutativity conditions are used to model the idea that the execution of certain instructions does not affect the result of certain tests. In light of this, the complexity of the universal Horn theory of KA and KAT are of interest. There are both positive and negative results. It is shown in [Kozen, 1997c] that for a Horn formula $\Phi \to \varphi$ over *-continuous Kleene algebras,

- if Φ contains only commutativity conditions $\alpha\beta = \beta\alpha$, the universal Horn theory is Π^0_1-complete;

- if Φ contains only monoid equations, the problem is Π^0_2-complete;
- for arbitrary finite sets of equations Φ, the problem is Π^1_1-complete.

On the other hand, commutativity assumptions of the form $\alpha\varphi = \varphi\alpha$, where φ is a test, and assumptions of the form $\gamma = 0$ can be eliminated without loss of efficiency, as shown in [Cohen, 1994a; Kozen and Smith, 1996]. Note that assumptions of this form are all we need to encode Hoare Logic as described above.

In typed Kleene algebra introduced in [Kozen, 1998; Kozen, 1999b], elements have types $s \to t$. This allows Kleene algebras of nonsquare matrices, among other applications. It is shown in [Kozen, 1999b] that Hoare Logic is subsumed by the type calculus of typed KA augmented with a typecast or coercion rule for tests. Thus Hoare-style reasoning with partial correctness assertions reduces to typechecking in a relatively simple type system.

14.6 Dynamic Algebra

Dynamic algebra provides an abstract algebraic framework that relates to PDL as Boolean algebra relates to propositional logic. A *dynamic algebra* is defined to be any two-sorted algebraic structure $(K,\ B,\ \cdot)$, where $B = (B,\ \to,\ 0)$ is a Boolean algebra, $K = (K,\ +,\ \cdot,\ {}^*,\ 0,\ 1)$ is a Kleene algebra (see Section 14.5), and $\cdot : K \times B \to B$ is a scalar multiplication satisfying algebraic constraints corresponding to the dual forms of the PDL axioms (Axioms 17). For example, all dynamic algebras satisfy the equations

$$\begin{aligned} (\alpha\beta) \cdot \varphi &= \alpha \cdot (\beta \cdot \varphi) \\ \alpha \cdot 0 &= 0 \\ 0 \cdot \varphi &= 0 \\ \alpha \cdot (\varphi \vee \psi) &= \alpha \cdot \varphi \vee \alpha \cdot \psi, \end{aligned}$$

which correspond to the PDL validities

$$\begin{aligned} \langle\alpha\,;\,\beta\rangle\varphi &\leftrightarrow \langle\alpha\rangle\langle\beta\rangle\varphi \\ \langle\alpha\rangle\mathbf{0} &\leftrightarrow \mathbf{0} \\ \langle\mathbf{0}?\rangle\varphi &\leftrightarrow \mathbf{0} \\ \langle\alpha\rangle(\varphi \vee \psi) &\leftrightarrow \langle\alpha\rangle\varphi \vee \langle\alpha\rangle\psi, \end{aligned}$$

respectively. The Boolean algebra B is an abstraction of the formulas of PDL and the Kleene algebra K is an abstraction of the programs.

The interaction of scalar multiplication with iteration can be axiomatized in a finitary or infinitary way. One can postulate

$$\alpha^* \cdot \varphi \quad\leq\quad \varphi \vee (\alpha^* \cdot (\neg\varphi \wedge (\alpha \cdot \varphi))) \tag{46}$$

corresponding to the diamond form of the PDL induction axiom (Axiom 17(viii)). Here $\varphi \leq \psi$ in B iff $\varphi \vee \psi = \psi$. Alternatively, one can postulate the stronger axiom of *-*continuity*:

$$\alpha^* \cdot \varphi \quad = \quad \sup_n (\alpha^n \cdot \varphi). \tag{47}$$

We can think of (47) as a conjunction of infinitely many axioms $\alpha^n \cdot \varphi \leq \alpha^* \cdot \varphi$, $n \geq 0$, and the infinitary Horn formula

$$(\bigwedge_{n \geq 0} \alpha^n \cdot \varphi \ \leq \ \psi) \quad \rightarrow \quad \alpha^* \cdot \varphi \ \leq \ \psi.$$

In the presence of the other axioms, (47) implies (46) ([Kozen, 1980b]), and is strictly stronger in the sense that there are dynamic algebras that are not *-continuous ([Pratt, 1979a]).

A standard Kripke frame $\mathfrak{K} = (U,\ \mathfrak{m}_{\mathfrak{K}})$ of PDL gives rise to a *-continuous dynamic algebra consisting of a Boolean algebra of subsets of U and a Kleene algebra of binary relations on U. Operators are interpreted as in PDL, including 0 as $\mathbf{0}$? (the empty program), 1 as $\mathbf{1}$? (the identity program), and $\alpha \cdot \varphi$ as $<\alpha>\varphi$. Nonstandard Kripke frames (see Section 3.2) also give rise to dynamic algebras, but not necessarily *-continuous ones. A dynamic algebra is *separable* if any pair of distinct Kleene elements can be distinguished by some Boolean element; that is, if $\alpha \neq \beta$, then there exists $\varphi \in B$ with $\alpha \cdot \varphi \neq \beta \cdot \varphi$.

Research directions in this area include the following.

- *Representation theory*. It is known that any separable dynamic algebra is isomorphic to some possibly nonstandard Kripke frame. Under certain conditions, "possibly nonstandard" can be replaced by "standard," but not in general, even for *-continuous algebras ([Kozen, 1980b; Kozen, 1979c; Kozen, 1980a]).
- *Algebraic methods in* PDL. The small model property (Theorem 15) and completeness (Theorem 18) for PDL can be established by purely algebraic considerations ([Pratt, 1980a]).
- *Comparative study of alternative axiomatizations of* *. For example, it is known that separable dynamic algebras can be distinguished from standard Kripke frames by a first-order formula, but even $L_{\omega_1\omega}$ cannot distinguish the latter from *-continuous separable dynamic algebras ([Kozen, 1981b]).
- *Equational theory of dynamic algebras*. Many seemingly unrelated models of computation share the same equational theory, namely that of dynamic algebras ([Pratt, 1979b; Pratt, 1979a]).

In addition, many interesting questions arise from the algebraic viewpoint, and interesting connections with topology, classical algebra, and model theory have been made ([Kozen, 1979b; Németi, 1980]).

15 BIBLIOGRAPHICAL NOTES

Systematic program verification originated with the work of [Floyd, 1967] and [Hoare, 1969]. Hoare Logic was introduced in [Hoare, 1969]; see [Cousot, 1990; Apt, 1981; Apt and Olderog, 1991] for surveys.

The *digital abstraction*, the view of computers as state transformers that operate by performing a sequence of discrete and instantaneous primitive steps, can be attributed to [Turing, 1936]. Finite-state transition systems were defined formally by [McCulloch and Pitts, 1943]. State-transition semantics is based on this idea and is quite prevalent in early work on program semantics and verification; see [Hennessy and Plotkin, 1979]. The relational-algebraic approach taken here, in which programs are interpreted as binary input/output relations, was introduced in the context of DL by [Pratt, 1976].

The notions of partial and total correctness were present in the early work of [Hoare, 1969]. Regular programs were introduced by [Fischer and Ladner, 1979] in the context of PDL. The concept of nondeterminism was introduced in the original paper of [Turing, 1936], although he did not develop the idea. Nondeterminism was further developed by [Rabin and Scott, 1959] in the context of finite automata.

[Burstall, 1974] suggested using modal logic for reasoning about programs, but it was not until the work of [Pratt, 1976], prompted by a suggestion of R. Moore, that it was actually shown how to extend modal logic in a useful way by considering a separate modality for every program. The first research devoted to propositional reasoning about programs seems to be that of [Fischer and Ladner, 1977; Fischer and Ladner, 1979] on PDL. As mentioned in the Preface, the general use of logical systems for reasoning about programs was suggested by [Engeler, 1967].

Other semantics besides Kripke semantics have been studied; see [Berman, 1979; Nishimura, 1979; Kozen, 1979b; Trnkova and Reiterman, 1980; Kozen, 1980b; Pratt, 1979b]. Modal logic has many applications and a vast literature; good introductions can be found in [Hughes and Cresswell, 1968; Chellas, 1980]. Alternative and iterative guarded commands were studied in [Gries, 1981]. Partial correctness assertions and the Hoare rules given in Section 2.6 were first formulated by [Hoare, 1969]. Regular expressions, on which the regular program operators are based, were introduced by [Kleene, 1956]. Their algebraic theory was further investigated by [Conway, 1971]. They were first applied in the context of DL by [Fischer and Ladner, 1977; Fischer and Ladner, 1979]. The axiomatization of PDL given in Axioms 17 was formulated by [Segerberg, 1977]. Tests and converse were investigated by various authors; see [Peterson, 1978; Berman, 1978; Berman and Paterson, 1981; Streett, 1981; Streett, 1982; Vardi, 1985b]. The continuity of the diamond operator in the presence of reverse is due to [Trnkova and Reiterman, 1980].

The filtration argument and the small model property for PDL are due to [Fischer and Ladner, 1977; Fischer and Ladner, 1979]. Nonstandard Kripke frames for PDL were studied by [Berman, 1979; Berman, 1982], [Parikh, 1978a], [Pratt, 1979a; Pratt, 1980a], and [Kozen, 1979c; Kozen, 1979b; Kozen, 1980a; Kozen, 1980b; Kozen, 1981b].

The axiomatization of PDL used here (Axiom System 17) was introduced by [Segerberg, 1977]. Completeness was shown independently by [Gabbay, 1977] and [Parikh, 1978a]. A short and easy-to-follow proof is given in [Kozen and Parikh, 1981]. Completeness is also treated in [Pratt, 1978; Pratt, 1980a; Berman, 1979; Nishimura, 1979; Kozen, 1981a].

The exponential-time lower bound for PDL was established by [Fischer and Ladner, 1977; Fischer and Ladner, 1979] by showing how PDL formulas can encode computations of linear-space-bounded alternating Turing machines.

Deterministic exponential-time algorithms were first given in [Pratt, 1978; Pratt, 1979b; Pratt, 1980b].

Theorem 24 showing that the problem of deciding whether $\Gamma \models \psi$, where Γ is a fixed r.e. set of PDL formulas, is Π^1_1-complete is due to [Meyer *et al.*, 1981].

The computational difficulty of the validity problem for nonregular PDL and the borderline between the decidable and undecidable were discussed in [Harel *et al.*, 1983]. The fact that any nonregular program adds expressive power to PDL, Theorem 25, first appeared explicitly in [Harel and Singerman, 1996]. Theorem 26 on the undecidability of context-free PDL was observed by [Ladner, 1977].

Theorems 27 and 28 are from [Harel *et al.*, 1983]. An alternative proof of Theorem 28 using tiling is supplied in [Harel, 1985]; see [Harel *et al.*, 2000]. The existence of a primitive recursive one-letter extension of PDL that is undecidable was shown already in [Harel *et al.*, 1983], but undecidability for the particular case of a^{2^*}, Theorem 29, is from [Harel and Paterson, 1984]. Theorem 30 is from [Harel and Singerman, 1996].

As to decidable extensions, Theorem 31 was proved in [Koren and Pnueli, 1983]. The more general results of Section 6.2, namely Theorems 32, 33, and 34, are from [Harel and Raz, 1993], as is the notion of a simple-minded PDA. The decidability of emptiness for pushdown and stack automata on trees that is needed for the proofs of these is from [Harel and Raz, 1994]. A better bound on the complexity of the emptiness results can be found in [Peng and Iyer, 1995].

A sufficient condition for PDL with the addition of a program over a single letter alphabet not to have the finite model property is given in [Harel and Singerman, 1996].

Completeness and exponential time decidability for DPDL, Theorem 40 and the upper bound of Theorem 41, are proved in [Ben-Ari *et al.*, 1982] and [Valiev, 1980]. The lower bound of Theorem 41 is from [Parikh, 1981]. Theorems 43 and 44 on SDPDL are from [Halpern and Reif, 1981; Halpern and Reif, 1983].

That tests add to the power of PDL is proved in [Berman and Paterson, 1981]. It is also known that the test-depth hierarchy is strict [Berman, 1978; Peterson, 1978] and that rich-test PDL is strictly more expressive than poor-test PDL [Peterson, 1978; Berman, 1978; Berman and Paterson, 1981]. These results also hold for SDPDL.

The results on programs as automata (Theorems 45 and 46) appear in [Pratt, 1981b]. Alternative proofs are given in [Harel and Sherman, 1985]; see [Harel *et al.*, 2000]. In recent years, the development of the automata-theoretic approach to logics of programs has prompted renewed inquiry into the complexity of automata on infinite objects, with considerable success. See [Courcoubetis and Yannakakis, 1988; Emerson, 1985; Emerson and Jutla, 1988; Emerson and Sistla, 1984; Manna and Pnueli, 1987; Muller *et al.*, 1988; Pecuchet, 1986; Safra, 1988; Sistla *et al.*, 1987; Streett, 1982; Vardi, 1985a; Vardi, 1985b; Vardi, 1987; Vardi and Stockmeyer, 1985; Vardi and Wolper, 1986b; Vardi and Wolper, 1986a; Arnold, 1997a; Arnold, 1997b]; and [Thomas, 1997]. Especially noteworthy in this area is the result of [Safra, 1988] involving the complexity of converting a nondeterministic automaton on infinite strings into an equivalent deterministic one. This result has already had a significant impact on the complexity of decision procedures for several logics of programs; see [Courcoubetis and Yannakakis, 1988; Emerson and Jutla, 1988; Emerson and Jutla, 1989]; and [Safra, 1988].

Intersection of programs was studied in [Harel *et al.*, 1982a]. That the axioms for converse yield completeness for CPDL is proved in [Parikh, 1978a].

The complexity of PDL with converse and various forms of well-foundedness constructs is studied in [Vardi, 1985b]. Many authors have studied logics with a least-fixpoint operator, both on the propositional and first-order levels ([Scott and de Bakker, 1969; Hitchcock and Park, 1972; Park, 1976; Pratt, 1981a; Kozen, 1982; Kozen, 1983; Kozen, 1988; Kozen and Parikh, 1983; Niwinski, 1984; Streett, 1985a; Vardi and Stockmeyer, 1985]). The version of the propositional μ-calculus presented here was introduced in [Kozen, 1982; Kozen, 1983].

That the propositional μ-calculus is strictly more expressive than PDL with **wf** was show in [Niwinski, 1984] and [Streett, 1985a]. That this logic is strictly more expressive than PDL with **halt** was shown in [Harel and Sherman, 1982]. That this logic is strictly more expressive than PDL was shown in [Streett, 1981].

The **wf** construct (actually its complement, **repeat**) is investigated in [Streett, 1981; Streett, 1982], in which Theorems 48 (which is actually due to Pratt) and 50–52 are proved. The **halt** construct (actually its complement, **loop**) was introduced in [Harel and Pratt, 1978] and Theorem 49 is from [Harel and Sherman, 1982]. Finite model properties for the logics LPDL, RPDL, CLPDL, CRPDL, and the propositional μ-calculus were established in [Streett, 1981; Streett, 1982] and [Kozen, 1988]. Decidability results were obtained in [Streett, 1981; Streett, 1982; Kozen and Parikh, 1983; Vardi and Stockmeyer, 1985]; and [Vardi, 1985b]. Deterministic exponential-time completeness was established in [Emerson and Jutla, 1988] and [Safra, 1988]. For the strongest variant, CRPDL, exponential-time decidability follows from [Vardi, 1998b].

Concurrent PDL is defined and studied in [Peleg, 1987b]. Additional versions of this logic, which employ various mechanisms for communication among the concurrent parts of a program, are considered in [Peleg, 1987c; Peleg, 1987a]. These papers contain many results concerning expressive power, decidability and undecidability for concurrent PDL with communication.

Other work on PDL not described here includes work on nonstandard models, studied in [Berman, 1979; Berman, 1982] and [Parikh, 1981]; PDL with Boolean assignments, studied in [Abrahamson, 1980]; and restricted forms of the consequence problem, studied in [Parikh, 1981].

First-order DL was defined in [Harel *et al.*, 1977], where it was also first named Dynamic Logic. That paper was carried out as a direct continuation of the original work of [Pratt, 1976].

Many variants of DL were defined in [Harel, 1979]. In particular, DL(bstk) is very close to the context-free Dynamic Logic investigated there.

Uninterpreted reasoning in the form of program schematology has been a common activity ever since the work of [Ianov, 1960]. It was given considerable impetus by the work of [Luckham *et al.*, 1970] and [Paterson and Hewitt, 1970]; see also [Greibach, 1975]. The study of the correctness of interpreted programs goes back to the work of Turing and von Neumann, but seems to have become a well-defined area of research following [Floyd, 1967], [Hoare, 1969] and [Manna, 1974].

Embedding logics of programs in $L_{\omega_1\omega}$ is based on observations of [Engeler, 1967]. Theorem 57 is from [Meyer and Parikh, 1981]. Theorem 60 is from [Harel, 1979] (see also [Harel, 1984] and [Harel and Kozen, 1984]); it is similar to the expressiveness result of [Cook, 1978]. Theorem 61 and Corollary 62 are from [Harel and Kozen, 1984].

Arithmetical structures were first defined by [Moschovakis, 1974] under the name *acceptable structures*. In the context of logics of programs, they were reintroduced and studied in [Harel, 1979].

The Π^1_1-completeness of DL was first proved by Meyer, and Theorem 63 appears in [Harel *et al.*, 1977]. An alternative proof is given in [Harel, 1985]; see [Harel *et al.*, 2000]. Theorem 65 is from [Meyer and Halpern, 1982]. That the fragment of DL considered in Theorem 66 is not r.e., was proved by [Pratt, 1976]. Theorem 67 follows from [Harel and Kozen, 1984].

The name “spectral complexity” was proposed by [Tiuryn, 1986], although the main ideas and many results concerning this notion were already present in [Tiuryn and Urzyczyn, 1983] (see [Tiuryn and Urzyczyn, 1988] for the full version). This notion is an instance of the so-called *second-order spectrum* of a formula. First-order spectra were investigated by [Sholz, 1952], from which originates the well known *Spectralproblem*. The reader can find more about this problem and related results in the survey paper by [Börger, 1984]. The notion of a natural chain is from [Urzyczyn, 1983]. The results presented here are from [Tiuryn and Urzyczyn, 1983; Tiuryn and Urzyczyn, 1988]. A result similar to Theorem 69 in the area of finite model theory was obtained by [Sazonov, 1980] and independently by [Gurevich, 1983]. Higher-order stacks were introduced in [Engelfriet, 1983] to study complexity classes. Higher-order arrays and stacks in DL were considered by [Tiuryn, 1986], where a strict hierarchy within the class of elementary recursive sets was established. The main tool used in the proof of the strictness of this hierarchy is a generalization of Cook’s auxiliary pushdown automata theorem for higher-order stacks, which is due to [Kowalczyk *et al.*, 1987].

[Meyer and Halpern, 1982] showed completeness for termination assertions (Theorem 71). Infinitary completeness for DL (Theorem 72) is based upon a similar result for Algorithmic Logic (see Section 13.1) by [Mirkowska, 1971]. The proof sketch presented in [Harel *et al.*, 2000] is an adaptation of Henkin's proof for $L_{\omega_1 \omega}$ appearing in [Keisler, 1971].

The notion of relative completeness and Theorem 73 are due to [Cook, 1978]. The notion of arithmetical completeness and Theorem 75 is from [Harel, 1979].

The use of invariants to prove partial correctness and of well-founded sets to prove termination are due to [Floyd, 1967]. An excellent survey of such methods and the corresponding completeness results appears in [Apt, 1981].

Some contrasting negative results are contained in [Clarke, 1979], [Lipton, 1977], and [Wand, 1978].

Many of the results on relative expressiveness presented herein answer questions posed in [Harel, 1979]. Similar uninterpreted research, comparing the expressive power of classes of programs (but detached from any surrounding logic) has taken place under the name *comparative schematology* quite extensively ever since [Ianov, 1960]; see [Greibach, 1975] and [Manna, 1974].

Theorems 76, 79 and 83(i) result as an application of the so-called *spectral theorem*, which connects expressive power of logics with complexity classes. This theorem was obtained by [Tiuryn and Urzyczyn, 1983; Tiuryn and Urzyczyn, 1984; Tiuryn and Urzyczyn, 1988]. A simplified framework for this approach and a statement of this theorem together with a proof is given in [Harel *et al.*, 2000].

Theorem 78 appears in [Berman *et al.*, 1982] and was proved independently in [Stolboushkin and Taitslin, 1983]. An alternative proof is given in [Tiuryn, 1989]. These results extend in a substantial way an earlier and much simpler result for the case of regular programs without equality in the vocabulary, which appears in [Halpern, 1981]. A simpler proof of the special case of the quantifier-free fragment of the logic of regular programs appears in [Meyer and Winklmann, 1982]. Theorem 79 is from [Tiuryn and Urzyczyn, 1984].

Theorem 80 is from [Stolboushkin, 1983]. The proof, as in the case of regular programs (see [Stolboushkin and Taitslin, 1983]), uses Adian's result from group theory ([Adian, 1979]). Results on the expressive power of DL with deterministic **while** programs and a Boolean stack can be found in [Stolboushkin, 1983; Kfoury, 1985]. Theorem 81 is from [Tiuryn and Urzyczyn, 1983; Tiuryn and Urzyczyn, 1988].

[Erimbetov, 1981; Tiuryn, 1981b; Tiuryn, 1984; Kfoury, 1983; Kfoury and Stolboushkin, 1997] contain results on the expressive power of DL over programming languages with bounded memory. [Erimbetov, 1981] shows that $\mathsf{DL}(\mathrm{dreg}) < \mathsf{DL}(\mathrm{dstk})$. The main proof technique is pebble games on finite trees.

Theorem 83 is from [Urzyczyn, 1987]. There is a different proof of this result, using Adian structures, which appears in [Stolboushkin, 1989]. Theorem 77 is from [Urzyczyn, 1988], which also studies programs with Boolean arrays.

Wildcard assignments were considered in [Harel *et al.*, 1977] under the name *nondeterministic assignments*. Theorem 84 is from [Meyer and Winklmann, 1982].

Theorem 85 is from [Meyer and Parikh, 1981].

In our exposition of the comparison of the expressive power of logics, we have made the assumption that programs use only quantifier-free first-order tests. It follows from the results of [Urzyczyn, 1986] that allowing full first-order tests in many cases results in increased expressive power. [Urzyczyn, 1986] also proves that adding array assignments to nondeterministic r.e. programs increases the expressive power of the logic. This should be contrasted with the result of [Meyer and Tiuryn, 1981; Meyer and Tiuryn, 1984] to the effect that for deterministic r.e. programs, array assignments do not increase expressive power.

[Makowski, 1980] considers a weaker notion of equivalence between logics common in investigations in abstract model theory, whereby models are extended with interpretations for additional predicate symbols. With this notion it is shown in [Makowski, 1980] that most of the versions of logics of programs treated here become equivalent.

Algorithmic logic was introduced by [Salwicki, 1970]. [Mirkowska, 1980; Mirkowska, 1981a; Mirkowska, 1981b] extended AL to allow nondeterministic **while** programs and studied the operators ∇ and Δ. Complete infinitary deductive systems for propositional and first-order versions were given by [Mirkowska, 1980; Mirkowska, 1981a; Mirkowska, 1981b] using the algebraic methods of [Rasiowa and Sikorski, 1963]. Surveys of early work in AL can be found in [Banachowski *et al.*, 1977; Salwicki, 1977]. [Constable, 1977; Constable and O'Donnell, 1978; Goldblatt, 1982] presented logics similar to AL and DL for reasoning about deterministic **while** programs.

Nonstandard Dynamic Logic was introduced by [Németi, 1981] and [Andréka *et al.*, 1982a; Andréka *et al.*, 1982b] and studied in [Csirmaz, 1985]. See [Makowski and Sain, 1986] for more information and further references.

The **halt** construct (actually its complement, **loop**) was introduced in [Harel and Pratt, 1978], and the **wf** construct (actually its complement, **repeat**) was investigated for PDL in [Streett, 1981; Streett, 1982]. Theorem 86 is from [Meyer and Winklmann, 1982], Theorem 87 is from [Harel and Peleg, 1985], Theorem 88 is from [Harel, 1984], and the axiomatizations of LDL and PDL are discussed in [Harel, 1979; Harel, 1984].

Dynamic algebra was introduced in [Kozen, 1980b] and [Pratt, 1979b] and studied by numerous authors; see [Kozen, 1979c; Kozen, 1979b; Kozen, 1980a; Kozen, 1981b; Pratt, 1979a; Pratt, 1980a; Pratt, 1988; Németi, 1980; Trnkova and Reiterman, 1980]. A survey of the main results appears in [Kozen, 1979a].

The PhD thesis of [Ramshaw, 1981] contains an engaging introduction to the subject of probabilistic semantics and verification. [Kozen, 1981d] provided a formal semantics for probabilistic programs. The logic $Pr(\mathsf{DL})$ was presented in [Feldman and Harel, 1984], along with a deductive system that is complete for Kozen's semantics relative to an extension of first-order analysis. Various propositional versions of probabilistic DL have been proposed in [Reif, 1980; Makowsky and Tiomkin, 1980; Feldman, 1984; Parikh and Mahoney, 1983; Kozen, 1985]. The temporal approach to probabilistic verification has been studied in [Lehmann and Shelah, 1982; Hart *et al.*, 1982; Courcoubetis and Yannakakis, 1988; Vardi, 1985a]. Interest in the subject of probabilistic verification has undergone a recent revival; see [Morgan *et al.*, 1999; Segala and Lynch, 1994; Hansson and Jonsson, 1994; Jou and Smolka, 1990; Baier and Kwiatkowska, 1998; Huth and Kwiatkowska, 1997; Blute *et al.*, 1997].

Concurrent DL is defined and studied in [Peleg, 1987b]. Additional versions of this logic, which employ various mechanisms for communication among the concurrent parts of a program, are also considered in [Peleg, 1987c; Peleg, 1987a].

David Harel
The Weizmann Institute of Science, Rehovot, Israel

Dexter Kozen
Cornell University, Ithaca, New York

Jerzy Tiuryn
The University of Warsaw, Warsaw, Poland

BIBLIOGRAPHY

[Abrahamson, 1980] K. Abrahamson. *Decidability and expressiveness of logics of processes*. PhD thesis, Univ. of Washington, 1980.

[Adian, 1979] S. I. Adian. *The Burnside Problem and Identities in Groups*. Springer-Verlag, 1979.

[Aho *et al.*, 1975] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1975.

[Ambler *et al.*, 1995] S. Ambler, M. Kwiatkowska, and N. Measor. Duality and the completeness of the modal μ-calculus. *Theor. Comput. Sci.*, 151(1):3–27, November 1995.

[Andréka *et al.*, 1982a] H. Andréka, I. Németi, and I. Sain. A complete logic for reasoning about programs via nonstandard model theory, part I. *Theor. Comput. Sci.*, 17:193–212, 1982.

[Andréka *et al.*, 1982b] H. Andréka, I. Németi, and I. Sain. A complete logic for reasoning about programs via nonstandard model theory, part II. *Theor. Comput. Sci.*, 17:259–278, 1982.

[Apt and Olderog, 1991] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

[Apt and Plotkin, 1986] K. R. Apt and G. Plotkin. Countable nondeterminism and random assignment. *J. Assoc. Comput. Mach.*, 33:724–767, 1986.

[Apt, 1981] K. R. Apt. Ten years of Hoare's logic: a survey—part I. *ACM Trans. Programming Languages and Systems*, 3:431–483, 1981.

[Archangelsky, 1992] K. V. Archangelsky. A new finite complete solvable quasiequational calculus for algebra of regular languages. Manuscript, Kiev State University, 1992.

[Arnold, 1997a] A. Arnold. An initial semantics for the μ-calculus on trees and Rabin's complementation lemma. Technical report, University of Bordeaux, 1997.

[Arnold, 1997b] A. Arnold. The μ-calculus on trees and Rabin's complementation theorem. Technical report, University of Bordeaux, 1997.

[Backhouse, 1975] Roland Carl Backhouse. *Closure Algorithms and the Star-Height Problem of Regular Languages*. PhD thesis, Imperial College, London, U.K., 1975.

[Backhouse, 1986] R. C. Backhouse. *Program Construction and Verification*. Prentice-Hall, 1986.

[Baier and Kwiatkowska, 1998] Christel Baier and Marta Kwiatkowska. On the verification of qualitative properties of probabilistic processes under fairness constraints. *Information Processing Letters*, 66(2):71–79, April 1998.

[Banachowski *et al.*, 1977] L. Banachowski, A. Kreczmar, G. Mirkowska, H. Rasiowa, and A. Salwicki. An introduction to algorithmic logic: metamathematical investigations in the theory of programs. In Mazurkiewitz and Pawlak, editors, *Math. Found. Comput. Sci.*, pages 7–99. Banach Center, Warsaw, 1977.

[Ben-Ari *et al.*, 1982] M. Ben-Ari, J. Y. Halpern, and A. Pnueli. Deterministic propositional dynamic logic: finite models, complexity and completeness. *J. Comput. Syst. Sci.*, 25:402–417, 1982.

[Berman and Paterson, 1981] F. Berman and M. Paterson. Propositional dynamic logic is weaker without tests. *Theor. Comput. Sci.*, 16:321–328, 1981.

[Berman *et al.*, 1982] P. Berman, J. Y. Halpern, and J. Tiuryn. On the power of nondeterminism in dynamic logic. In Nielsen and Schmidt, editors, *Proc 9th Colloq. Automata Lang. Prog.*, volume 140 of *Lect. Notes in Comput. Sci.*, pages 48–60. Springer-Verlag, 1982.

[Berman, 1978] F. Berman. Expressiveness hierarchy for PDL with rich tests. Technical Report 78-11-01, Comput. Sci. Dept., Univ. of Washington, 1978.

[Berman, 1979] F. Berman. A completeness technique for D-axiomatizable semantics. In *Proc. 11th Symp. Theory of Comput.*, pages 160–166. ACM, 1979.

[Berman, 1982] F. Berman. Semantics of looping programs in propositional dynamic logic. *Math. Syst. Theory*, 15:285–294, 1982.

[Bhat and Cleaveland, 1996] Girish Bhat and Rance Cleaveland. Efficient local model checking for fragments of the modal μ-calculus. In T. Margaria and B. Steffen, editors, *Proc. Second Int. Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lect. Notes in Comput. Sci.*, pages 107–112. Springer-Verlag, March 1996.

[Bloom and Ésik, 1992] Stephen L. Bloom and Zoltán Ésik. Program correctness and matricial iteration theories. In *Proc. 7th Int. Conf. Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 457–476. Springer-Verlag, 1992.

[Bloom and Ésik, 1993] Stephen L. Bloom and Zoltán Ésik. Equational axioms for regular sets. *Math. Struct. Comput. Sci.*, 3:1–24, 1993.

[Blute *et al.*, 1997] R. Blute, J. Desharnais, A. Edelat, and P. Panangaden. Bisimulation for labeled Markov processes. In *Proc. 12th Symp. Logic in Comput. Sci.*, pages 149–158. IEEE, 1997.

[Boffa, 1990] Maurice Boffa. Une remarque sur les systèmes complets d'identités rationnelles. *Informatique Théoretique et Applications/Theoretical Informatics and Applications*, 24(4):419–423, 1990.

[Boffa, 1995] Maurice Boffa. Une condition impliquant toutes les identités rationnelles. *Informatique Théoretique et Applications/Theoretical Informatics and Applications*, 29(6):515–518, 1995.

[Bonsangue and Kwiatkowska, 1995] M. Bonsangue and M. Kwiatkowska. Re-interpreting the modal μ-calculus. In A. Ponse, M. van Rijke, and Y. Venema, editors, *Modal Logic and Process Algebra*, pages 65–83. CSLI Lecture Notes, August 1995.

[Börger, 1984] E. Börger. Spectralproblem and completeness of logical decision problems. In G. Hasenjaeger E. Börger and D. Rödding, editors, *Logic and Machines: Decision Problems and Complexity, Proccedings*, volume 171 of *Lect. Notes in Comput. Sci.*, pages 333–356. Springer-Verlag, 1984.

[Bradfield, 1996] Julian C. Bradfield. The modal μ-calculus alternation hierarchy is strict. In U. Montanari and V. Sassone, editors, *Proc. CONCUR'96*, volume 1119 of *Lect. Notes in Comput. Sci.*, pages 233–246. Springer, 1996.

[Burstall, 1974] R. M. Burstall. Program proving as hand simulation with a little induction. *Information Processing*, pages 308–312, 1974.

[Chandra *et al.*, 1981] Ashok Chandra, Dexter Kozen, and Larry Stockmeyer. Alternation. *J. Assoc. Comput. Mach.*, 28(1):114–133, 1981.

[Chellas, 1980] B. F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, 1980.

[Clarke, 1979] E. M. Clarke. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *J. Assoc. Comput. Mach.*, 26:129–147, 1979.

[Cleaveland, 1996] Rance Cleaveland. Efficient model checking via the equational μ-calculus. In *Proc. 11th Symp. Logic in Comput. Sci.*, pages 304–312. IEEE, July 1996.

[Cohen *et al.*, 1996] Ernie Cohen, Dexter Kozen, and Frederick Smith. The complexity of Kleene algebra with tests. Technical Report 96-1598, Computer Science Department, Cornell University, July 1996.

[Cohen, 1994a] Ernie Cohen. Hypotheses in Kleene algebra. Available as ftp://ftp.telcordia.com/pub/ernie/research/homepage.html, April 1994.

[Cohen, 1994b] Ernie Cohen. Lazy caching. Available as ftp://ftp.telcordia.com/pub/ernie/research/homepage.html, 1994.

[Cohen, 1994c] Ernie Cohen. Using Kleene algebra to reason about concurrency control. Available as ftp://ftp.telcordia.com/pub/ernie/research/homepage.html, 1994.

[Constable and O'Donnell, 1978] R. L. Constable and M. O'Donnell. *A Programming Logic*. Winthrop, 1978.

[Constable, 1977] R. L. Constable. On the theory of programming logics. In *Proc. 9th Symp. Theory of Comput.*, pages 269–285. ACM, May 1977.

[Conway, 1971] John Horton Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[Cook, 1978] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7:70–80, 1978.

[Courcoubetis and Yannakakis, 1988] C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *Proc. 29th Symp. Foundations of Comput. Sci.*, pages 338–345. IEEE, October 1988.

[Cousot, 1990] Patrick Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbood of Theoretical Computer Science*, volume B, pages 841–993. Elsevier, Amsterdam, 1990.

[Csirmaz, 1985] L. Csirmaz. A completeness theorem for dynamic logic. *Notre Dame J. Formal Logic*, 26:51–60, 1985.

[Davis *et al.*, 1994] M. D. Davis, R. Sigal, and E. J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, 1994.

[de Bakker, 1980] J. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.

[Emerson and Halpern, 1985] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *J. Comput. Syst. Sci.*, 30(1):1–24, 1985.

[Emerson and Halpern, 1986] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: on branching vs. linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[Emerson and Jutla, 1988] E. A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th Symp. Foundations of Comput. Sci.*, pages 328–337. IEEE, October 1988.

[Emerson and Jutla, 1989] E. A. Emerson and C. Jutla. On simultaneously determinizing and complementing ω-automata. In *Proc. 4th Symp. Logic in Comput. Sci.* IEEE, June 1989.

[Emerson and Lei, 1986] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proc. 1st Symp. Logic in Comput. Sci.*, pages 267–278. IEEE, June 1986.

[Emerson and Lei, 1987] E. A. Emerson and C. L. Lei. Modalities for model checking: branching time strikes back. *Sci. Comput. Programming*, 8:275–306, 1987.

[Emerson and Sistla, 1984] E. A. Emerson and P. A. Sistla. Deciding full branching-time logic. *Infor. and Control*, 61:175–201, 1984.

[Emerson, 1985] E. A. Emerson. Automata, tableax, and temporal logics. In R. Parikh, editor, *Proc. Workshop on Logics of Programs*, volume 193 of *Lect. Notes in Comput. Sci.*, pages 79–88. Springer-Verlag, 1985.

[Emerson, 1990] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of theoretical computer science*, volume B: formal models and semantics, pages 995–1072. Elsevier, 1990.

[Engeler, 1967] E. Engeler. Algorithmic properties of structures. *Math. Syst. Theory*, 1:183–195, 1967.

[Engelfriet, 1983] J. Engelfriet. Iterated pushdown automata and complexity classes. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 365–373, Boston, Massachusetts, 1983.

[Erimbetov, 1981] M. M. Erimbetov. On the expressive power of programming logics. In *Proc. Alma-Ata Conf. Research in Theoretical Programming*, pages 49–68, 1981. In Russian.

[Feldman and Harel, 1984] Y. A. Feldman and D. Harel. A probabilistic dynamic logic. *J. Comput. Syst. Sci.*, 28:193–215, 1984.

[Feldman, 1984] Y. A. Feldman. A decidable propositional dynamic logic with explicit probabilities. *Infor. and Control*, 63:11–38, 1984.

[Fischer and Ladner, 1977] Michael J. Fischer and Richard E. Ladner. Propositional modal logic of programs. In *Proc. 9th Symp. Theory of Comput.*, pages 286–294. ACM, 1977.

[Fischer and Ladner, 1979] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

[Floyd, 1967] R. W. Floyd. Assigning meanings to programs. In *Proc. Symp. Appl. Math.*, volume 19, pages 19–31. AMS, 1967.

[Friedman, 1971] H. Friedman. Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory. In Gandy and Yates, editors, *Logic Colloq. 1969*, pages 361–390. North-Holland, 1971.

[Gabbay *et al.*, 1980] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. 7th Symp. Princip. Prog. Lang.*, pages 163–173. ACM, 1980.

[Gabbay *et al.*, 1994] D. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*. Oxford University Press, 1994.

[Gabbay, 1977] D. Gabbay. Axiomatizations of logics of programs. Unpublished, 1977.

[Goldblatt, 1982] R. Goldblatt. *Axiomatising the Logic of Computer Programming*, volume 130 of *Lect. Notes in Comput. Sci.* Springer-Verlag, 1982.

[Goldblatt, 1987] R. Goldblatt. Logics of time and computation. Technical Report Lect. Notes 7, Center for the Study of Language and Information, Stanford Univ., 1987.

[Greibach, 1975] S. Greibach. *Theory of Program Structures: Schemes, Semantics, Verification*, volume 36 of *Lecture Notes in Computer Science*. Springer Verlag, 1975.

[Gries, 1981] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[Gurevich, 1983] Yu. Gurevich. Algebras of feasible functions. In *24-th IEEE Annual Symposium on Foundations of Computer Science*, pages 210–214, 1983.

[Halpern and Reif, 1981] J. Y. Halpern and J. H. Reif. The propositional dynamic logic of deterministic, well-structured programs. In *Proc. 22nd Symp. Found. Comput. Sci.*, pages 322–334. IEEE, 1981.

[Halpern and Reif, 1983] J. Y. Halpern and J. H. Reif. The propositional dynamic logic of deterministic, well-structured programs. *Theor. Comput. Sci.*, 27:127–165, 1983.

[Halpern, 1981] J. Y. Halpern. On the expressive power of dynamic logic II. Technical Report TM-204, MIT/LCS, 1981.

[Halpern, 1982] J. Y. Halpern. Deterministic process logic is elementary. In *Proc. 23rd Symp. Found. Comput. Sci.*, pages 204–216. IEEE, 1982.

[Halpern, 1983] J. Y. Halpern. Deterministic process logic is elementary. *Infor. and Control*, 57(1):56–89, 1983.

[Hansson and Jonsson, 1994] H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6:512–535, 1994.

[Harel and Kozen, 1984] David Harel and Dexter Kozen. A programming language for the inductive sets, and applications. *Information and Control*, 63(1–2):118–139, 1984.

[Harel and Paterson, 1984] D. Harel and M. S. Paterson. Undecidability of PDL with $L = \{a^{2^i} \mid i \geq 0\}$. *J. Comput. Syst. Sci.*, 29:359–365, 1984.

[Harel and Peleg, 1985] D. Harel and D. Peleg. More on looping vs. repeating in dynamic logic. *Information Processing Letters*, 20:87–90, 1985.

[Harel and Pratt, 1978] D. Harel and V. R. Pratt. Nondeterminism in logics of programs. In *Proc. 5th Symp. Princip. Prog. Lang.*, pages 203–213. ACM, 1978.

[Harel and Raz, 1993] D. Harel and D. Raz. Deciding properties of nonregular programs. *SIAM J. Comput.*, 22:857–874, 1993.

[Harel and Raz, 1994] D. Harel and D. Raz. Deciding emptiness for stack automata on infinite trees. *Information and Computation*, 113:278–299, 1994.

[Harel and Sherman, 1982] D. Harel and R. Sherman. Looping vs. repeating in dynamic logic. *Infor. and Control*, 55:175–192, 1982.

[Harel and Sherman, 1985] D. Harel and R. Sherman. Propositional dynamic logic of flowcharts. *Infor. and Control*, 64:119–135, 1985.

[Harel and Singerman, 1996] D. Harel and E. Singerman. More on nonregular PDL: Finite models and Fibonacci-like programs. *Information and Computation*, 128:109–118, 1996.

[Harel *et al.*, 1977] D. Harel, A. R. Meyer, and V. R. Pratt. Computability and completeness in logics of programs. In *Proc. 9th Symp. Theory of Comput.*, pages 261–268. ACM, 1977.

[Harel *et al.*, 1982a] D. Harel, A. Pnueli, and M. Vardi. Two dimensional temporal logic and PDL with intersection. Unpublished, 1982.

[Harel *et al.*, 1982b] David Harel, Dexter Kozen, and Rohit Parikh. Process logic: Expressiveness, decidability, completeness. *J. Comput. Syst. Sci.*, 25(2):144–170, 1982.

[Harel *et al.*, 1983] D. Harel, A. Pnueli, and J. Stavi. Propositional dynamic logic of nonregular programs. *J. Comput. Syst. Sci.*, 26:222–243, 1983.

[Harel *et al.*, 2000] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, MA, 2000.

[Harel, 1979] D. Harel. *First-Order Dynamic Logic*, volume 68 of *Lect. Notes in Comput. Sci.* Springer-Verlag, 1979.

[Harel, 1984] D. Harel. Dynamic logic. In Gabbay and Guenthner, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, pages 497–604. Reidel, 1984.

[Harel, 1985] D. Harel. Recurring dominoes: Making the highly undecidable highly understandable. *Annals of Discrete Mathematics*, 24:51–72, 1985.

[Harel, 1992] D. Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, second edition, 1992.

[Hart *et al.*, 1982] S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. In *Proc. 9th Symp. Princip. Prog. Lang.*, pages 1–6. ACM, 1982.

[Hartonas, 1998] Chrysafis Hartonas. Duality for modal μ-logics. *Theor. Comput. Sci.*, 202(1–2):193–222, 1998.

[Hennessy and Plotkin, 1979] M. C. B. Hennessy and G. D. Plotkin. Full abstraction for a simple programming language. In *Proc. Symp. Semantics of Algorithmic Languages*, volume 74 of *Lecture Notes in Computer Science*, pages 108–120. Springer-Verlag, 1979.

[Hitchcock and Park, 1972] P. Hitchcock and D. Park. Induction rules and termination proofs. In M. Nivat, editor, *Int. Colloq. Automata Lang. Prog.*, pages 225–251. North-Holland, 1972.

[Hoare, 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. Assoc. Comput. Mach.*, 12:576–580, 583, 1969.

[Hopkins and Kozen, 1999] Mark Hopkins and Dexter Kozen. Parikh's theorem in commutative Kleene algebra. In *Proc. Conf. Logic in Computer Science (LICS'99)*, pages 394–401. IEEE, July 1999.

[Hughes and Cresswell, 1968] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic*. Methuen, 1968.

[Huth and Kwiatkowska, 1997] M. Huth and M. Kwiatkowska. Quantitative analysis and model checking. In *Proc. 12th Symp. Logic in Comput. Sci.*, pages 111–122. IEEE, 1997.

[Ianov, 1960] Y. I. Ianov. The logical schemes of algorithms. In *Problems of Cybernetics*, volume 1, pages 82–140. Pergamon Press, 1960.

[Iwano and Steiglitz, 1990] Kazuo Iwano and Kenneth Steiglitz. A semiring on convex polygons and zero-sum cycle problems. *SIAM J. Comput.*, 19(5):883–901, 1990.

[Jou and Smolka, 1990] C. Jou and S. Smolka. Equivalences, congruences and complete axiomatizations for probabilistic processes. In *Proc. CONCUR'90*, volume 458 of *Lecture Notes in Comput. Sci.*, pages 367–383. Springer-Verlag, 1990.

[Kaivola, 1997] R. Kaivola. *Using Automata to Characterise Fixed Point Temporal Logics*. PhD thesis, University of Edinburgh, April 1997. Report CST-135-97.

[Kamp, 1968] H. W. Kamp. *Tense logics and the theory of linear order*. PhD thesis, UCLA, 1968.

[Keisler, 1971] J. Keisler. *Model Theory for Infinitary Logic*. North Holland, 1971.

[Kfoury and Stolboushkin, 1997] A.J. Kfoury and A.P. Stolboushkin. An infinite pebble game and applications. *Information and Computation*, 136:53–66, 1997.

[Kfoury, 1983] A.J. Kfoury. Definability by programs in first-order structures. *Theoretical Computer Science*, 25:1–66, 1983.

[Kfoury, 1985] A. J. Kfoury. Definability by deterministic and nondeterministic programs with applications to first-order dynamic logic. *Infor. and Control*, 65(2–3):98–121, 1985.

[Kleene, 1956] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, N.J., 1956.

[Knijnenburg, 1988] P. M. W. Knijnenburg. On axiomatizations for propositional logics of programs. Technical Report RUU-CS-88-34, Rijksuniversiteit Utrecht, November 1988.

[Koren and Pnueli, 1983] T. Koren and A. Pnueli. There exist decidable context-free propositional dynamic logics. In *Proc. Symp. on Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 290–312. Springer-Verlag, 1983.

[Kowalczyk *et al.*, 1987] W. Kowalczyk, D. Niwiński, and J. Tiuryn. A generalization of Cook's auxiliary–pushdown–automata theorem. *Fundamenta Informaticae*, XII:497–506, 1987.

[Kozen and Parikh, 1981] Dexter Kozen and Rohit Parikh. An elementary proof of the completeness of *PDL. Theor. Comput. Sci.*, 14(1):113–118, 1981.

[Kozen and Parikh, 1983] Dexter Kozen and Rohit Parikh. A decision procedure for the propositional μ-calculus. In Clarke and Kozen, editors, *Proc. Workshop on Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 313–325. Springer-Verlag, 1983.

[Kozen and Patron, 2000] Dexter Kozen and Maria-Cristina Patron. Certification of compiler optimizations using Kleene algebra with tests. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luis Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proc. 1st Int. Conf. Computational Logic (CL2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 568–582, London, July 2000. Springer-Verlag.

[Kozen and Smith, 1996] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In D. van Dalen and M. Bezem, editors, *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259, Utrecht, The Netherlands, September 1996. Springer-Verlag.

[Kozen and Tiuryn, 1990] Dexter Kozen and Jerzy Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 789–840. North Holland, Amsterdam, 1990.

[Kozen, 1979a] Dexter Kozen. Dynamic algebra. In E. Engeler, editor, *Proc. Workshop on Logic of Programs*, volume 125 of *Lecture Notes in Computer Science*, pages 102–144. Springer-Verlag, 1979. chapter of *Propositional dynamic logics of programs: A survey* by Rohit Parikh.

[Kozen, 1979b] Dexter Kozen. On the duality of dynamic algebras and Kripke models. In E. Engeler, editor, *Proc. Workshop on Logic of Programs*, volume 125 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, 1979.

[Kozen, 1979c] Dexter Kozen. On the representation of dynamic algebras. Technical Report RC7898, IBM Thomas J. Watson Research Center, October 1979.

[Kozen, 1980a] Dexter Kozen. On the representation of dynamic algebras II. Technical Report RC8290, IBM Thomas J. Watson Research Center, May 1980.

[Kozen, 1980b] Dexter Kozen. A representation theorem for models of *-free *PDL*. In *Proc. 7th Colloq. Automata, Languages, and Programming*, pages 351–362. EATCS, July 1980.

[Kozen, 1981a] Dexter Kozen. Logics of programs. Lecture notes, Aarhus University, Denmark, 1981.

[Kozen, 1981b] Dexter Kozen. On induction vs. *-continuity. In Kozen, editor, *Proc. Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 167–176, New York, 1981. Springer-Verlag.

[Kozen, 1981c] Dexter Kozen. On the expressiveness of μ. Manuscript, 1981.

[Kozen, 1981d] Dexter Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22:328–350, 1981.

[Kozen, 1982] Dexter Kozen. Results on the propositional μ-calculus. In *Proc. 9th Int. Colloq. Automata, Languages, and Programming*, pages 348–359, Aarhus, Denmark, July 1982. EATCS.

[Kozen, 1983] Dexter Kozen. Results on the propositional μ-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[Kozen, 1984] Dexter Kozen. A Ramsey theorem with infinitely many colors. In Lenstra, Lenstra, and van Emde Boas, editors, *Dopo Le Parole*, pages 71–72. University of Amsterdam, Amsterdam, May 1984.

[Kozen, 1985] Dexter Kozen. A probabilistic *PDL*. *J. Comput. Syst. Sci.*, 30(2):162–178, April 1985.

[Kozen, 1988] Dexter Kozen. A finite model theorem for the propositional μ-calculus. *Studia Logica*, 47(3):233–241, 1988.

[Kozen, 1990] Dexter Kozen. On Kleene algebras and closed semirings. In Rovan, editor, *Proc. Math. Found. Comput. Sci.*, volume 452 of *Lecture Notes in Computer Science*, pages 26–47, Banska-Bystrica, Slovakia, 1990. Springer-Verlag.

[Kozen, 1991a] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. In *Proc. 6th Symp. Logic in Comput. Sci.*, pages 214–225, Amsterdam, July 1991. IEEE.

[Kozen, 1991b] Dexter Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, New York, 1991.

[Kozen, 1994a] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.

[Kozen, 1994b] Dexter Kozen. On action algebras. In J. van Eijck and A. Visser, editors, *Logic and Information Flow*, pages 78–88. MIT Press, 1994.

[Kozen, 1996] Dexter Kozen. Kleene algebra with tests and commutativity conditions. In T. Margaria and B. Steffen, editors, *Proc. Second Int. Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 14–33, Passau, Germany, March 1996. Springer-Verlag.

[Kozen, 1997a] Dexter Kozen. *Automata and Computability*. Springer-Verlag, New York, 1997.

[Kozen, 1997b] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[Kozen, 1997c] Dexter Kozen. On the complexity of reasoning in Kleene algebra. In *Proc. 12th Symp. Logic in Comput. Sci.*, pages 195–202, Los Alamitos, Ca., June 1997. IEEE.

[Kozen, 1998] Dexter Kozen. Typed Kleene algebra. Technical Report 98-1669, Computer Science Department, Cornell University, March 1998.

[Kozen, 1999a] Dexter Kozen. On Hoare logic and Kleene algebra with tests. In *Proc. Conf. Logic in Computer Science (LICS'99)*, pages 167–172. IEEE, July 1999.

[Kozen, 1999b] Dexter Kozen. On Hoare logic, Kleene algebra, and types. Technical Report 99-1760, Computer Science Department, Cornell University, July 1999. Abstract in: Abstracts of 11th Int. Congress Logic, Methodology and Philosophy of Science, Ed. J. Cachro and K. Kijania-Placek, Krakow, Poland, August 1999, p. 15. To appear in: Proc. 11th Int. Congress Logic, Methodology and Philosophy of Science, ed. P. Gardenfors, K. Kijania-Placek and J. Wolenski, Kluwer.

[Krob, 1991] Daniel Krob. A complete system of B-rational identities. *Theoretical Computer Science*, 89(2):207–343, October 1991.

[Kuich and Salomaa, 1986] Werner Kuich and Arto Salomaa. *Semirings, Automata, and Languages*. Springer-Verlag, Berlin, 1986.

[Kuich, 1987] Werner Kuich. The Kleene and Parikh theorem in complete semirings. In T. Ottmann, editor, *Proc. 14th Colloq. Automata, Languages, and Programming*, volume 267 of *Lecture Notes in Computer Science*, pages 212–225, New York, 1987. EATCS, Springer-Verlag.

[Ladner, 1977] R. E. Ladner. Unpublished, 1977.

[Lamport, 1980] L. Lamport. "Sometime" is sometimes "not never". *Proc. 7th Symp. Princip. Prog. Lang.*, pages 174–185, 1980.

[Lehmann and Shelah, 1982] D. Lehmann and S. Shelah. Reasoning with time and chance. *Infor. and Control*, 53(3):165–198, 1982.

[Lewis and Papadimitriou, 1981] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1981.

[Lipton, 1977] R. J. Lipton. A necessary and sufficient condition for the existence of Hoare logics. In *Proc. 18th Symp. Found. Comput. Sci.*, pages 1–6. IEEE, 1977.

[Luckham *et al.*, 1970] D. C. Luckham, D. Park, and M. Paterson. On formalized computer programs. *J. Comput. Syst. Sci.*, 4:220–249, 1970.

[Mader, 1997] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. PhD thesis, Fakultt fr Informatik, Technische Universitt Mnchen, September 1997.

[Makowski and Sain, 1986] J. A. Makowski and I. Sain. On the equivalence of weak second-order and nonstandard time semantics for various program verification systems. In *Proc. 1st Symp. Logic in Comput. Sci.*, pages 293–300. IEEE, 1986.

[Makowski, 1980] J. A. Makowski. Measuring the expressive power of dynamic logics: an application of abstract model theory. In *Proc. 7th Int. Colloq. Automata Lang. Prog.*, volume 80 of *Lect. Notes in Comput. Sci.*, pages 409–421. Springer-Verlag, 1980.

[Makowsky and Tiomkin, 1980] J. A. Makowsky and M. L. Tiomkin. Probabilistic propositional dynamic logic. Manuscript, 1980.

[Manna and Pnueli, 1981] Z. Manna and A. Pnueli. Verification of concurrent programs: temporal proof principles. In D. Kozen, editor, *Proc. Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comput. Sci.*, pages 200–252. Springer-Verlag, 1981.

[Manna and Pnueli, 1987] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by $\forall$-automata. In *Proc. 14th Symp. Principles of Programming Languages*, pages 1–12. ACM, January 1987.

[Manna, 1974] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[McCulloch and Pitts, 1943] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophysics*, 5:115–143, 1943.

[Mehlhorn, 1984] K. Mehlhorn. *Graph Algorithms and NP-Completeness*, volume II of *Data Structures and Algorithms*. Springer-Verlag, 1984.

[Meyer and Halpern, 1982] A. R. Meyer and J. Y. Halpern. Axiomatic definitions of programming languages: a theoretical assessment. *J. Assoc. Comput. Mach.*, 29:555–576, 1982.

[Meyer and Parikh, 1981] A. R. Meyer and R. Parikh. Definability in dynamic logic. *J. Comput. Syst. Sci.*, 23:279–298, 1981.

[Meyer and Tiuryn, 1981] A. R. Meyer and J. Tiuryn. A note on equivalences among logics of programs. In D. Kozen, editor, *Proc. Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comput. Sci.*, pages 282–299. Springer-Verlag, 1981.

[Meyer and Tiuryn, 1984] A. R. Meyer and J. Tiuryn. Equivalences among logics of programs. *Journal of Computer and Systems Science*, 29:160–170, 1984.

[Meyer and Winklmann, 1982] A. R. Meyer and K. Winklmann. Expressing program looping in regular dynamic logic. *Theor. Comput. Sci.*, 18:301–323, 1982.

[Meyer *et al.*, 1981] A. R. Meyer, R. S. Streett, and G. Mirkowska. The deducibility problem in propositional dynamic logic. In E. Engeler, editor, *Proc. Workshop Logic of Programs*, volume 125 of *Lect. Notes in Comput. Sci.*, pages 12–22. Springer-Verlag, 1981.

[Miller, 1976] G. L. Miller. Riemann's hypothesis and tests for primality. *J. Comput. Syst. Sci.*, 13:300–317, 1976.

[Mirkowska, 1971] G. Mirkowska. On formalized systems of algorithmic logic. *Bull. Acad. Polon. Sci. Ser. Sci. Math. Astron. Phys.*, 19:421–428, 1971.

[Mirkowska, 1980] G. Mirkowska. Algorithmic logic with nondeterministic programs. *Fund. Informaticae*, III:45–64, 1980.

[Mirkowska, 1981a] G. Mirkowska. PAL—propositional algorithmic logic. In E. Engeler, editor, *Proc. Workshop Logic of Programs*, volume 125 of *Lect. Notes in Comput. Sci.*, pages 23–101. Springer-Verlag, 1981.

[Mirkowska, 1981b] G. Mirkowska. PAL—propositional algorithmic logic. *Fund. Informaticae*, IV:675–760, 1981.

[Morgan *et al.*, 1999] C. Morgan, A. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Trans. Programming Languages and Systems*, 8(1):1–30, 1999.

[Moschovakis, 1974] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, 1974.

[Muller *et al.*, 1988] D. E. Muller, A. Saoudi, and P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proc. 3rd Symp. Logic in Computer Science*, pages 422–427. IEEE, July 1988.

[Németi, 1980] I. Németi. Every free algebra in the variety generated by the representable dynamic algebras is separable and representable. Manuscript, 1980.

[Németi, 1981] I. Németi. Nonstandard dynamic logic. In D. Kozen, editor, *Proc. Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comput. Sci.*, pages 311–348. Springer-Verlag, 1981.

[Ng and Tarski, 1977] K. C. Ng and A. Tarski. Relation algebras with transitive closure, abstract 742-02-09. *Notices Amer. Math. Soc.*, 24:A29–A30, 1977.

[Ng, 1984] K. C. Ng. *Relation Algebras with Transitive Closure*. PhD thesis, University of California, Berkeley, 1984.

[Nishimura, 1979] H. Nishimura. Sequential method in propositional dynamic logic. *Acta Informatica*, 12:377–400, 1979.

[Nishimura, 1980] H. Nishimura. Descriptively complete process logic. *Acta Informatica*, 14:359–369, 1980.

[Niwinski, 1984] D. Niwinski. The propositional μ-calculus is more expressive than the propositional dynamic logic of looping. University of Warsaw, 1984.

[Parikh and Mahoney, 1983] R. Parikh and A. Mahoney. A theory of probabilistic programs. In E. Clarke and D. Kozen, editors, *Proc. Workshop on Logics of Programs*, volume 164 of *Lect. Notes in Comput. Sci.*, pages 396–402. Springer-Verlag, 1983.

[Parikh, 1978a] R. Parikh. The completeness of propositional dynamic logic. In *Proc. 7th Symp. on Math. Found. of Comput. Sci.*, volume 64 of *Lect. Notes in Comput. Sci.*, pages 403–415. Springer-Verlag, 1978.

[Parikh, 1978b] R. Parikh. A decidability result for second order process logic. In *Proc. 19th Symp. Found. Comput. Sci.*, pages 177–183. IEEE, 1978.

[Parikh, 1981] R. Parikh. Propositional dynamic logics of programs: a survey. In E. Engeler, editor, *Proc. Workshop on Logics of Programs*, volume 125 of *Lect. Notes in Comput. Sci.*, pages 102–144. Springer-Verlag, 1981.

[Parikh, 1983] Rohit Parikh. Propositional game logic. In *Proc. 23rd IEEE Symp. Foundations of Computer Science*, 1983.

[Park, 1976] D. Park. Finiteness is μ-ineffable. *Theor. Comput. Sci.*, 3:173–181, 1976.

[Paterson and Hewitt, 1970] M. S. Paterson and C. E. Hewitt. Comparative schematology. In *Record Project MAC Conf. on Concurrent Systems and Parallel Computation*, pages 119–128. ACM, 1970.

[Pecuchet, 1986] J. P. Pecuchet. On the complementation of Büchi automata. *Theor. Comput. Sci.*, 47:95–98, 1986.

[Peleg, 1987a] D. Peleg. Communication in concurrent dynamic logic. *J. Comput. Sys. Sci.*, 35:23–58, 1987.

[Peleg, 1987b] D. Peleg. Concurrent dynamic logic. *J. Assoc. Comput. Mach.*, 34(2):450–479, 1987.

[Peleg, 1987c] D. Peleg. Concurrent program schemes and their logics. *Theor. Comput. Sci.*, 55:1–45, 1987.

[Peng and Iyer, 1995] W. Peng and S. Purushothaman Iyer. A new type of pushdown-tree automata on infinite trees. *Int. J. of Found. of Comput. Sci.*, 6(2):169–186, 1995.

[Peterson, 1978] G. L. Peterson. The power of tests in propositional dynamic logic. Technical Report 47, Comput. Sci. Dept., Univ. of Rochester, 1978.

[Pnueli, 1977] A. Pnueli. The temporal logic of programs. In *Proc. 18th Symp. Found. Comput. Sci.*, pages 46–57. IEEE, 1977.

[Pratt, 1976] V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proc. 17th Symp. Found. Comput. Sci.*, pages 109–121. IEEE, 1976.

[Pratt, 1978] V. R. Pratt. A practical decision method for propositional dynamic logic. In *Proc. 10th Symp. Theory of Comput.*, pages 326–337. ACM, 1978.

[Pratt, 1979a] V. R. Pratt. Dynamic algebras: examples, constructions, applications. Technical Report TM-138, MIT/LCS, July 1979.

[Pratt, 1979b] V. R. Pratt. Models of program logics. In *Proc. 20th Symp. Found. Comput. Sci.*, pages 115–122. IEEE, 1979.

[Pratt, 1979c] V. R. Pratt. Process logic. In *Proc. 6th Symp. Princip. Prog. Lang.*, pages 93–100. ACM, 1979.

[Pratt, 1980a] V. R. Pratt. Dynamic algebras and the nature of induction. In *Proc. 12th Symp. Theory of Comput.*, pages 22–28. ACM, 1980.

[Pratt, 1980b] V. R. Pratt. A near-optimal method for reasoning about actions. *J. Comput. Syst. Sci.*, 20(2):231–254, 1980.

[Pratt, 1981a] V. R. Pratt. A decidable μ-calculus: preliminary report. In *Proc. 22nd Symp. Found. Comput. Sci.*, pages 421–427. IEEE, 1981.

[Pratt, 1981b] V. R. Pratt. Using graphs to understand PDL. In D. Kozen, editor, *Proc. Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comput. Sci.*, pages 387–396. Springer-Verlag, 1981.

[Pratt, 1988] Vaughan Pratt. Dynamic algebras as a well-behaved fragment of relation algebras. In D. Pigozzi, editor, *Proc. Conf. on Algebra and Computer Science*, volume 425 of *Lecture Notes in Computer Science*, pages 77–110, Ames, Iowa, June 1988. Springer-Verlag.

[Pratt, 1990] Vaughan Pratt. Action logic and pure induction. In J. van Eijck, editor, *Proc. Logics in AI: European Workshop JELIA '90*, volume 478 of *Lecture Notes in Computer Science*, pages 97–120, New York, September 1990. Springer-Verlag.

[Rabin and Scott, 1959] M. O. Rabin and D. S. Scott. Finite automata and their decision problems. *IBM J. Res. Develop.*, 3(2):115–125, 1959.

[Rabin, 1980] M. O. Rabin. Probabilistic algorithms for testing primality. *J. Number Theory*, 12:128–138, 1980.

[Ramshaw, 1981] L. H. Ramshaw. *Formalizing the analysis of algorithms*. PhD thesis, Stanford Univ., 1981.

[Rasiowa and Sikorski, 1963] H. Rasiowa and R. Sikorski. *Mathematics of Metamathematics*. Polish Scientific Publishers, PWN, 1963.

[Redko, 1964] V. N. Redko. On defining relations for the algebra of regular events. *Ukrain. Mat. Z.*, 16:120–126, 1964. In Russian.

[Reif, 1980] J. Reif. Logics for probabilistic programming. In *Proc. 12th Symp. Theory of Comput.*, pages 8–13. ACM, 1980.

[Rogers, 1967] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.

[Safra, 1988] S. Safra. On the complexity of ω-automata. In *Proc. 29th Symp. Foundations of Comput. Sci.*, pages 319–327. IEEE, October 1988.

[Sakarovitch, 1987] Jacques Sakarovitch. Kleene's theorem revisited: A formal path from Kleene to Chomsky. In A. Kelemenova and J. Keleman, editors, *Trends, Techniques, and Problems in Theoretical Computer Science*, volume 281 of *Lecture Notes in Computer Science*, pages 39–50, New York, 1987. Springer-Verlag.

[Salomaa, 1966] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. Assoc. Comput. Mach.*, 13(1):158–169, January 1966.

[Salwicki, 1970] A. Salwicki. Formalized algorithmic languages. *Bull. Acad. Polon. Sci. Ser. Sci. Math. Astron. Phys.*, 18:227–232, 1970.

[Salwicki, 1977] A. Salwicki. Algorithmic logic: a tool for investigations of programs. In Butts and Hintikka, editors, *Logic Foundations of Mathematics and Computability Theory*, pages 281–295. Reidel, 1977.

[Sazonov, 1980] V.Y. Sazonov. Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kibernetik*, 16:319–323, 1980.

[Scott and de Bakker, 1969] D. S. Scott and J. W. de Bakker. A theory of programs. IBM Vienna, 1969.

[Segala and Lynch, 1994] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In *Proc. CONCUR'94*, volume 836 of *Lecture Notes in Comput. Sci.*, pages 481–496. Springer-Verlag, 1994.

[Segerberg, 1977] K. Segerberg. A completeness theorem in the modal logic of programs (preliminary report). *Not. Amer. Math. Soc.*, 24(6):A–552, 1977.

[Shoenfield, 1967] J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[Sholz, 1952] H. Sholz. Ein ungelöstes Problem in der symbolischen Logik. *The Journal of Symbolic Logic*, 17:160, 1952.

[Sistla and Clarke, 1982] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. In *Proc. 14th Symp. Theory of Comput.*, pages 159–168. ACM, 1982.

[Sistla *et al.*, 1987] A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with application to temporal logic. *Theor. Comput. Sci.*, 49:217–237, 1987.

[Sokolsky and Smolka, 1994] O. Sokolsky and S. Smolka. Incremental model checking in the modal μ-calculus. In D. Dill, editor, *Proc. Conf. Computer Aided Verification*, volume 818 of *Lect. Notes in Comput. Sci.*, pages 352–363. Springer, June 1994.

[Steffen *et al.*, 1996] Bernhard Steffen, Tiziana Margaria, Andreas Classen, Volker Braun, Rita Nisius, and Manfred Reitenspiess. A constraint oriented service environment. In T. Margaria and B. Steffen, editors, *Proc. Second Int. Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lect. Notes in Comput. Sci.*, pages 418–421. Springer, March 1996.

[Stirling and Walker, 1989] C. Stirling and D. Walker. Local model checking in the modal μ-calculus. In *Proc. Int. Joint Conf. Theory and Practice of Software Develop. (TAPSOFT89)*, volume 352 of *Lect. Notes in Comput. Sci.*, pages 369–383. Springer, March 1989.

[Stirling, 1992] Colin Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 477–563. Clarendon Press, 1992.

[Stockmeyer and Meyer, 1973] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proc. 5th Symp. Theory of Computing*, pages 1–9, New York, 1973. ACM.

[Stolboushkin and Taitslin, 1983] A. P. Stolboushkin and M. A. Taitslin. Deterministic dynamic logic is strictly weaker than dynamic logic. *Infor. and Control*, 57:48–55, 1983.

[Stolboushkin, 1983] A.P. Stolboushkin. Regular dynamic logic is not interpretable in deterministic context-free dynamic logic. *Information and Computation*, 59:94–107, 1983.

[Stolboushkin, 1989] A.P. Stolboushkin. Some complexity bounds for dynamic logic. In *Proc. 4th Symp. Logic in Comput. Sci.*, pages 324–332. IEEE, June 1989.

[Streett and Emerson, 1984] Robert Streett and E. Allan Emerson. The propositional μ-calculus is elementary. In *Proc. 11th Int. Colloq. on Automata Languages and Programming*, pages 465–472. Springer, 1984. Lect. Notes in Comput. Sci. 172.

[Streett, 1981] R. S. Streett. Propositional dynamic logic of looping and converse. In *Proc. 13th Symp. Theory of Comput.*, pages 375–381. ACM, 1981.

[Streett, 1982] R. S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Infor. and Control*, 54:121–141, 1982.

[Streett, 1985a] R. S. Streett. Fixpoints and program looping: reductions from the propositional μ-calculus into propositional dynamic logics of looping. In R. Parikh, editor, *Proc. Workshop on Logics of Programs*, volume 193 of *Lect. Notes in Comput. Sci.*, pages 359–372. Springer-Verlag, 1985.

[Streett, 1985b] Robert Streett. Fixpoints and program looping: reductions from the propositional μ-calculus into propositional dynamic logics of looping. In Parikh, editor, *Proc. Workshop on Logics of Programs 1985*, pages 359–372. Springer, 1985. Lect. Notes in Comput. Sci. 193.

[Tarjan, 1981] Robert E. Tarjan. A unified approach to path problems. *J. Assoc. Comput. Mach.*, pages 577–593, 1981.

[Thiele, 1966] H. Thiele. Wissenschaftstheoretische untersuchungen in algorithmischen sprachen. In *Theorie der Graphschemata-Kalkale Veb Deutscher Verlag der Wissenschaften*. Berlin, 1966.

[Thomas, 1997] W. Thomas. Languages, automata, and logic. Technical Report 9607, Christian-Albrechts-Universität Kiel, May 1997.

[Tiuryn and Urzyczyn, 1983] J. Tiuryn and P. Urzyczyn. Some relationships between logics of programs and complexity theory. In *Proc. 24th Symp. Found. Comput. Sci.*, pages 180–184. IEEE, 1983.

[Tiuryn and Urzyczyn, 1984] J. Tiuryn and P. Urzyczyn. Remarks on comparing expressive power of logics of programs. In Chytil and Koubek, editors, *Proc. Math. Found. Comput. Sci.*, volume 176 of *Lect. Notes in Comput. Sci.*, pages 535–543. Springer-Verlag, 1984.

[Tiuryn and Urzyczyn, 1988] J. Tiuryn and P. Urzyczyn. Some relationships between logics of programs and complexity theory. *Theor. Comput. Sci.*, 60:83–108, 1988.

[Tiuryn, 1981a] J. Tiuryn. A survey of the logic of effective definitions. In E. Engeler, editor, *Proc. Workshop on Logics of Programs*, volume 125 of *Lect. Notes in Comput. Sci.*, pages 198–245. Springer-Verlag, 1981.

[Tiuryn, 1981b] J. Tiuryn. Unbounded program memory adds to the expressive power of first-order programming logics. In *Proc. 22nd Symp. Found. Comput. Sci.*, pages 335–339. IEEE, 1981.

[Tiuryn, 1984] J. Tiuryn. Unbounded program memory adds to the expressive power of first-order programming logics. *Infor. and Control*, 60:12–35, 1984.

[Tiuryn, 1986] J. Tiuryn. Higher-order arrays and stacks in programming: an application of complexity theory to logics of programs. In Gruska and Rovan, editors, *Proc. Math. Found. Comput. Sci.*, volume 233 of *Lect. Notes in Comput. Sci.*, pages 177–198. Springer-Verlag, 1986.

[Tiuryn, 1989] J. Tiuryn. A simplified proof of $\mathsf{DDL} < \mathsf{DL}$. *Information and Computation*, 81:1–12, 1989.

[Trnkova and Reiterman, 1980] V. Trnkova and J. Reiterman. Dynamic algebras which are not Kripke structures. In *Proc. 9th Symp. on Math. Found. Comput. Sci.*, pages 528–538, 1980.

[Turing, 1936] A. M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 42:230–265, 1936. Erratum: Ibid., 43 (1937), pp. 544–546.

[Urzyczyn, 1983] P. Urzyczyn. A necessary and sufficient condition in order that a Herbrand interpretation be expressive relative to recursive programs. *Information and Control*, 56:212–219, 1983.

[Urzyczyn, 1986] P. Urzyczyn. “During” cannot be expressed by “after”. *Journal of Computer and System Sciences*, 32:97–104, 1986.

[Urzyczyn, 1987] P. Urzyczyn. Deterministic context-free dynamic logic is more expressive than deterministic dynamic logic of regular programs. *Fundamenta Informaticae*, 10:123–142, 1987.

[Urzyczyn, 1988] P. Urzyczyn. Logics of programs with Boolean memory. *Fundamenta Informaticae*, XI:21–40, 1988.

[Valiev, 1980] M. K. Valiev. Decision complexity of variants of propositional dynamic logic. In *Proc. 9th Symp. Math. Found. Comput. Sci.*, volume 88 of *Lect. Notes in Comput. Sci.*, pages 656–664. Springer-Verlag, 1980.

[van Emde Boas, 1978] P. van Emde Boas. The connection between modal logic and algorithmic logics. In *Symp. on Math. Found. of Comp. Sci.*, pages 1–15, 1978.

[Vardi and Stockmeyer, 1985] M. Y. Vardi and L. Stockmeyer. Improved upper and lower bounds for modal logics of programs: preliminary report. In *Proc. 17th Symp. Theory of Comput.*, pages 240–251. ACM, May 1985.

[Vardi and Wolper, 1986a] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. Logic in Computer Science*, pages 332–344. IEEE, June 1986.

[Vardi and Wolper, 1986b] M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci.*, 32:183–221, 1986.

[Vardi, 1985a] M. Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proc. 26th Symp. Found. Comput. Sci.*, pages 327–338. IEEE, October 1985.

[Vardi, 1985b] M. Y. Vardi. The taming of the converse: reasoning about two-way computations. In R. Parikh, editor, *Proc. Workshop on Logics of Programs*, volume 193 of *Lect. Notes in Comput. Sci.*, pages 413–424. Springer-Verlag, 1985.

[Vardi, 1987] M. Y. Vardi. Verification of concurrent programs: the automata-theoretic framework. In *Proc. 2nd Symp. Logic in Comput. Sci.*, pages 167–176. IEEE, June 1987.

[Vardi, 1998a] M. Vardi. Linear vs. branching time: a complexity-theoretic perspective. In *Proc. 13th Symp. Logic in Comput. Sci.*, pages 394–405. IEEE, 1998.

[Vardi, 1998b] M. Y. Vardi. Reasoning about the past with two-way automata. In *Proc. 25th Int. Colloq. Automata Lang. Prog.*, volume 1443 of *Lect. Notes in Comput. Sci.*, pages 628–641. Springer-Verlag, July 1998.

[Walukiewicz, 1993] Igor Walukiewicz. Completeness result for the propositional μ-calculus. In *Proc. 8th IEEE Symp. Logic in Comput. Sci.*, June 1993.

[Walukiewicz, 1995] Igor Walukiewicz. Completeness of Kozen’s axiomatisation of the propositional μ-calculus. In *Proc. 10th Symp. Logic in Comput. Sci.*, pages 14–24. IEEE, June 1995.

[Walukiewicz, 2000] Igor Walukiewicz. Completeness of Kozen's axiomatisation of the propositional μ-calculus. *Infor. and Comput.*, 157(1–2):142–182, February–March 2000.

[Wand, 1978] M. Wand. A new incompleteness result for Hoare's system. *J. Assoc. Comput. Mach.*, 25:168–175, 1978.

[Wolper, 1981] P. Wolper. Temporal logic can be more expressive. In *Proc. 22nd Symp. Foundations of Computer Science*, pages 340–348. IEEE, 1981.

[Wolper, 1983] P. Wolper. Temporal logic can be more expressive. *Infor. and Control*, 56:72–99, 1983.